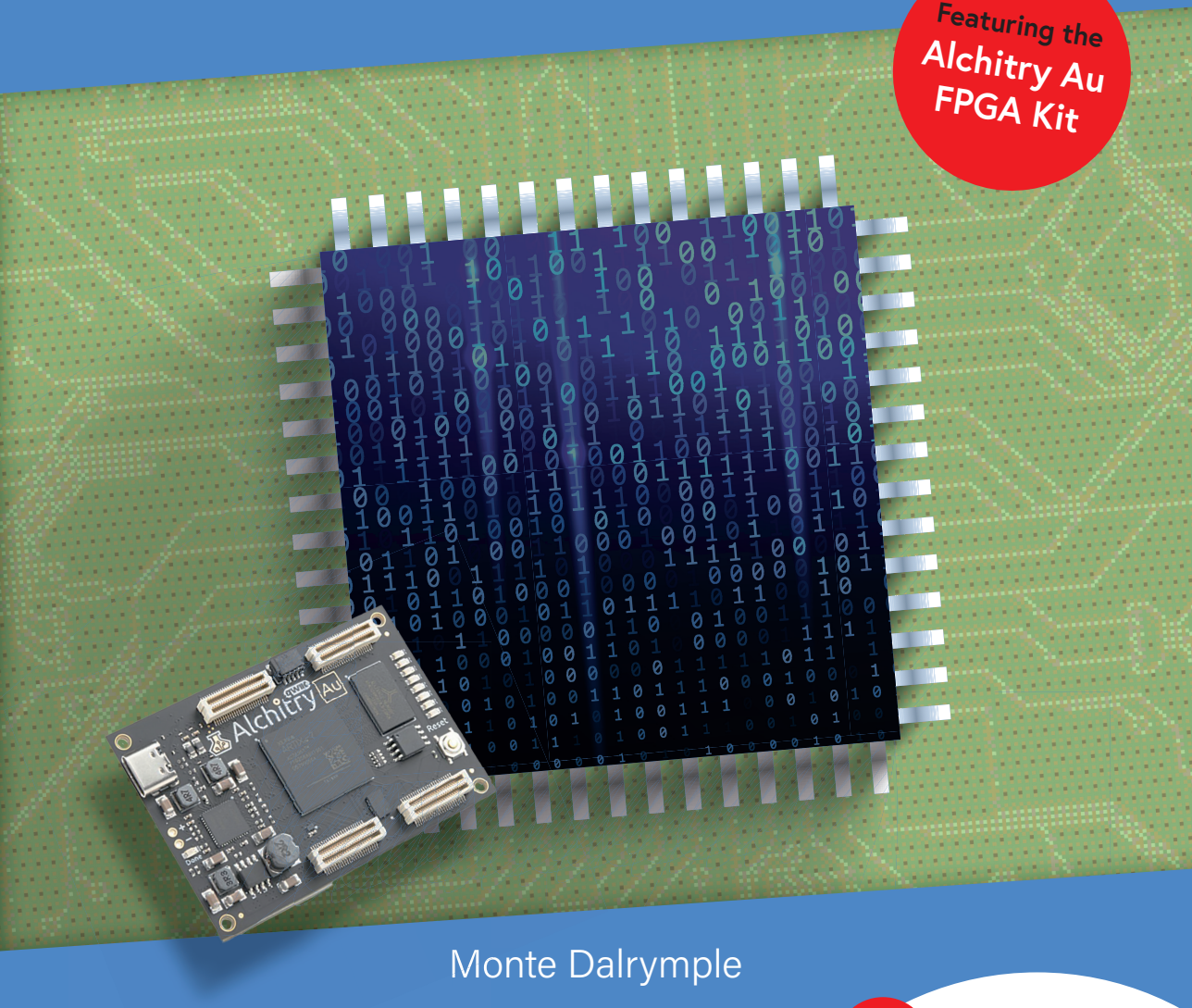
elektor books

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.

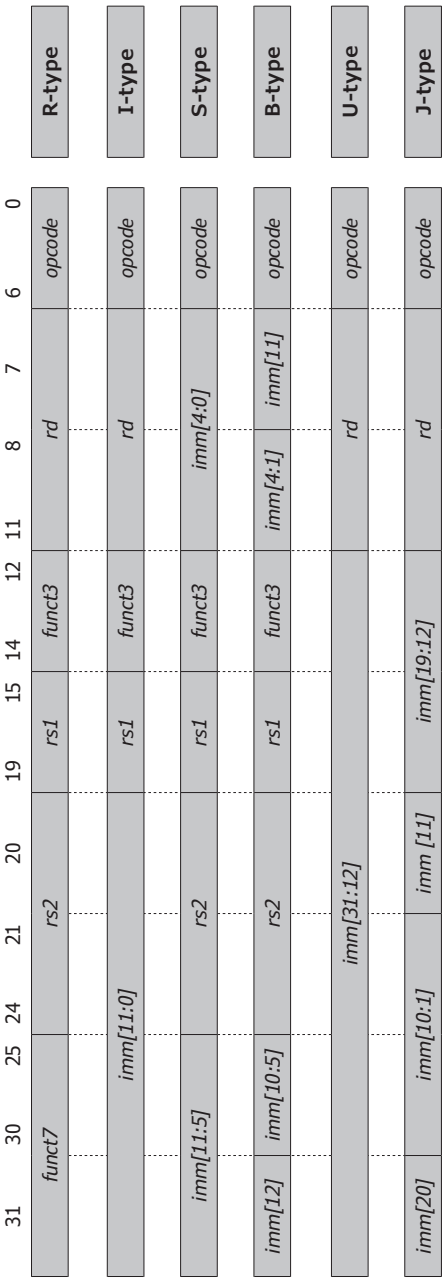


Figure 2.2: Instruction Formats.

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.