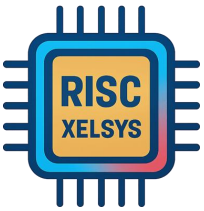
RISC-V Assembly Language and Architecture

May 26th 2026

Alan Johnson

Xelsys



Published by xelsys
www.risccomputing.com
info@alanjohnson.tech

Target Audience

- Embedded systems enthusiasts
- Computer architecture learners
- Developers interested in system-level programming
- Computer science students

Pre-requisites

Knowledge of the following areas will ease the journey.

- Familiarity with basic computer hardware

Microprocessor architecture

- Memory and data buses, registers, ALUs, …
- Experience with Linux ®
    - Installation of the operating system and applications
    - Bash
- Basic knowledge of the C programming language
- High school level math[1]
- A RISC-V system[2] or an emulated device.

[1] *Some of the optional tasks involve linear algebra, which will be more familiar to those at a higher level. A good reference is found at https://www.khanacademy.org/math/linear-algebra*

[2] *RISC-V based hardware is preferred over simulation.*

Overview

This document serves as a comprehensive, hands-on guide to RISC-V assembly language and architecture, specifically focusing on the unprivileged architecture. It is designed for computer science students, embedded systems enthusiasts, and developers seeking to master system-level programming.

Core Content Overview

The book balances foundational theory with practical application, covering:

- **Fundamentals:** Number systems (binary, hex, BCD, floating-point), logic operations, and the distinction between hardware, software, and firmware.
- **RISC-V Architecture:** Details on the **32/64-bit modes**, instruction extensions (M, F, D, C, V), and the **32 general-purpose registers** with their **ABI names**.
- **Programming Techniques:** Memory access (load/store), arithmetic/logic, control flow (branching/loops), stack management, and the use of **macros and functions**.
- **Integration:** Techniques for interfacing assembly with **C programming**, including inline assembly and calling conventions.
- **Advanced Processing:** Dedicated chapters on **Floating-Point (IEEE 754)** and **Vector operations (SIMD)**.
- **Toolchain & Simulation:** Practical guidance on using the **GNU toolchain** (assembler, linker, GDB, objdump, make) and simulators like **Spike**, **CPUlator**, and **RARS**.

Key Technical Data

- **Instruction Formats:** Covers the four base formats (**I-type, R-type, S-type, U-type**) plus **B-type** and **J-type** variants.
- **Memory & Addressing:** Explains **Little-Endian** storage, absolute/relative addressing, and **linker relaxation** (using the `.option norelax` directive).
- **Floating-Point:** Details single (32-bit) and double (64-bit) precision formats, including rounding modes and exception handling.
- **Vector Extensions:** Explains **LMUL** (register grouping), **SEW** (element width), and mask/tail attributes for parallel processing.

Notable Features

- **Practical Focus:** Chapters include code samples, step-by-step walkthroughs, and exercises.
- **Tooling:** Provides specific instructions for setting up environments on Linux, including configuring **QEMU-based virtual machines** and cross-compilation workflows.
- **Standardization:** Adheres to the **RISC-V ISA** standards, including the **Vector Extension 1.0**.
- **Formatting:** Uses clear typographical conventions, with computer input/output displayed in a courier **font** for readability.

This guide is intended to be a robust resource for both academic study and professional reference, providing the necessary knowledge to navigate the RISC-V ecosystem from bare-metal coding to high-level language integration.
The focus is on the RISC-V instruction set,unprivileged architecture.

Detailed Chapter Highlights

- **Chapter 1:** Lays the groundwork for assembly language, number systems, logic, and the rationale for using assembly.
- **Chapter 2:** Introduces RISC-V architecture, instruction set variants, register conventions, and essential tools (GNU assembler, linker, GDB, objdump, make).
- **Chapter 3:** Focuses on memory operations, addressing modes, linker relaxation, and practical examples of load/store instructions.
- **Chapter 4:** Explores arithmetic and logical operations, including overflow detection, multiplication/division, and shift instructions.
- **Chapter 5:** Covers control flow, loops, conditional and unconditional branching, and program structure.
- **Chapter 6:** Discusses stack management, modular code, macros, and function conventions.
- **Chapter 7:** Explains how to interface assembly with C, use inline assembly, and optimize code.
- **Chapter 8:** Details floating-point operations, IEEE 754 compliance, rounding modes, and exception handling.
- **Chapter 9:** Introduces vector processing, vector registers, SIMD operations, and advanced vector instructions.
- **Chapter 10:** Guides on cross-compiling, using the Spike simulator, and running/debugging RISC-V programs on various platforms.

Appendices and Resources

- **Appendices:** Include GDB commands, ASCII code tables, references, and assembly directives.
- **Figures, Listings, and Tables:** The book is rich with diagrams, code listings, and tables to illustrate concepts and provides a practical reference.

Typographical Conventions

Very few, -

- Narrative text uses the Calibri font.
- Key concepts are introduced with *italicized font.*
- Computer input and output text uses a `courier font with a light grey background.`

Conclusion

This document is a comprehensive, hands-on guide to RISC-V assembly language and architecture, suitable for learners and practitioners aiming to master low-level programming, system architecture, and the RISC-V ecosystem. It balances foundational theory with practical application, making it a valuable resource for both study and reference.

Contents

Figures

Tables

1 The Fundamentals of Assembly Language:

Overview of the chapter

Chapter 1 lays the foundation for understanding assembly language. The focus is on general principles that are essential before delving into the specifics of RISC-V. Topics include the purpose, structure, and advantages of assembly programming, and introduces the number systems and logic operations that underpin low-level code.

1.1 What is assembly language?

Assembly language is a computer language that is much closer to the operation of the computer itself. Today most coding is performed using languages that are easier for humans to understand. As far as assembly language goes the coding language uses *abbreviations* to give insight into the nature of the operation being performed. An example could be `bgt` which stands for *branch if greater than or less than* (some condition)

1.1.1 High-level languages vs. Assembly language

Many high-level languages place a strong emphasis on abstraction, treating functions as impenetrable black boxes and hides the inner workings. Over the last few decades, there has been a sharp rise in higher-level, object-oriented languages.

Assembly language takes a different approach from this abstraction and allows (indeed mandates) the coder to familiarize themselves with the inner workings of the system.

The former method is like a Rapid Application Development (RAD) methodology that works well with teams, whereas the second approach often involves smaller groups with specialized knowledge. Both approaches have their place. Digital computers inherently process data in one of two states (binary) so it is essential that we understand the low-level world of ones and zeros.

Surprisingly, assembly language coding has experienced a notable resurgence in popularity after an extended period of decline. This renewed interest can largely be attributed to the rapid progress in robotics, autonomous vehicles, and other self-governing technologies that critically depend on sensors reacting instantaneously to events. As a result, it is projected that the demand for skilled assembly language programmers will escalate significantly over the next ten years.

1.1.2 Architecture and Machine Code

Processors have different *architectures,* and each understands its own *machine code* instructions – their very heart, these instructions are combinations of binary numbers that instruct the processor how to proceed. Binary numbers are cumbersome for human operators, so instead, a set of *mnemonic* instructions is used. A hypothetical example could be an instruction such as `add r1, r2,r3` which would add two numbers together that are contained in register2[3] and register3, placing the result in register1 or `add r1, r2, 45` which would add the value 45 to the value contained in

[3] *Registers are low-capacity, high-speed storage elements, (typically anywhere from one to eight bytes in size) contained within the processor architecture.*

register2, placing the result in register1. The corresponding native machine code (again hypothetical) could be the binary code 10101100 00010010 00101100. The *mnemonic* instructions make up the *assembly language*.

1.1.3 Assembling, compiling and linking

The primary function of an `assembler` (a type of program) involves transforming assembly instructions, which are readable by programmers, into their equivalent machine code instructions. This generated code is referred to as an *object* file. Conversely, a `disassembler` performs the opposite task, converting machine code instructions back into assembly language. Additionally, the assembler fulfills other responsibilities, including comprehending a range of *directives* that allow for the definition and placement of data within the computer's memory addresses. For example, textual informational messages might be used to define error codes. These messages are established by the programmer, not by the specific processor itself. There are a number of these directives, and they will be discussed in more detail as the document progresses.

High-level programming languages utilize *compilers* to translate their code into machine instructions. Following the assembly or compilation phase, object files are *linked* together to construct an executable program. This *link*ing process can involve either single files or multiple files. Generally, instructions in high-level languages do not directly correspond one-to-one with the underlying machine code commands. They are crafted to be more intuitive for programmers, incorporating English-like keywords such as "`if...then,`" "`while,`" and "`print.`" High-level languages can either be *interprete*d, with machine code translation occurring during execution, or they can be precompiled into native machine code before runtime.

1.1.4 Pseudocode

Prior to developing genuine, syntactically accurate code, pseudocode finds its application. It serves to outline a series of algorithmic instructions, providing a high-level description of how a program will function. The chief advantage is enabling developers to concentrate on and organize upcoming tasks, thereby avoiding entanglement in minor syntactical details, even though conceptual mistakes may still arise. The common sequence of development is generally from an initial algorithm, → then to pseudocode, → and finally culminating in actual computer code. While pseudocode doesn't adhere to a rigid set of rules, it frequently employs keywords such as IF-THEN, WHILE, and GREATER THAN to delineate the program's execution path.

1.1.5 Why use assembly?

Assembly language maintains a close connection to its executing CPU, resulting in more optimized and efficient programs. Furthermore, it proves particularly adept for *system-level* programming tasks. However, a notable drawback arises from its verbosity, often requiring significantly more lines of code compared to high-level programming languages. This frequently leads to the adoption of a hybrid strategy: the main portion of the software can be developed in languages like *C* or Python, which then interact by passing and receiving da*t*a with specific, smalle*r* assembly code sections. Another concern is its lack of portability, given that assembly language is closely tied to the particular processor it operates on.

Programmers who specialize in assembly language frequently work with embedded systems and Internet of Things (IoT) devices. These systems commonly operate without an operating system and are consequently referred to as bare *metal.*

In the interests of education, this book will focus more on "pure" assembly coding rather than the pragmatic hybrid approach[4].

Proficient system-level programmers might consider bypassing this chapter, or perhaps just quickly reviewing it for a refresher. The topics explored wi*thin* are broad in nature and are not exclusively tailored to any *singular* system.

1.2 Hardware *vs Software Vs Firmware*

1.2.1 Hardware

In computer terms, *hardware* refers to the physical components that make up the system. Hardware is something that can be seen and touched.

1.2.2 Software

Software refers to the actual instructions that are loaded into the computer's memory. These instructions may direct the hardware to perform certain tasks. For example, system software is responsible for displaying the result of an operation onto a hardware output device, such as a display screen or printer, and for taking input from a device, such as a keyboard. In general, though, software is a set of instructions that causes an operation to occur, such as adding two numbers together.

1.2.2.1 Firmware

Firmware can be thought of as a set of instructions residing in hardware. This definition has become somewhat blurred, as these instructions were originally loaded onto read-only devices (ROMs). These devices would be physically replaced when new upgrade code was required. Over time, erasable programmable integrated circuits (IC's) (EPROMs) were introduced, which, as the name implies, could be written over with new code. Today, non-volatile random-access memory (NVRAM) devices are used and can often be upgraded on-line without even requiring a reboot. This process is sometimes referred to as flashing, since the underlying device is often flash memory.

1.3 Number Systems

Anthropologists may make a claim that we count in base 10 as this is the number of digits on our hands. Other cultures have used base 60 and base 20 (possibly using both fingers and toes). These number systems are not as well suited to computer systems, and today[5] base 2 and base 16 dominate when using low-level assembly programming.

1.3.1 Binary, Octal, Hexadecimal

Consider the base 10 number 4673_{10} – this breaks down into:

[4] *That is not to say that hybrid programming will be ignored within this text.*

[5] *Base 8 - Octal was also used on many earlier computers such as Digital Equipment Corporation's PDP family of minicomputers.*

4 x 1000
+
6 x 100
+
7 x 10
+
3 x 1
= 4000 + 600 + 70 + 3 = 4673

The use of ten (0-9) different characters, along with their positions, represented a major advance in computation when compared with systems such as the Roman counting method.

Digital electronic systems naturally gravitate towards a two-state binary system where current either flows or it does not. These two states are represented by the symbols 0 or 1.

Each individual binary digit is known as a *bit (b)*. For enhanced convenience, these binary digits are frequently clustered into groups of eight, forming what is termed a *Byte (B)*. Given that eight bits can express numerical values spanning from 00000000 to 11111111, their corresponding decimal values range from 0 to 255. A notable drawback of binary numbers is that representing a three-digit decimal number might necessitate as many as ten binary digits. A more space-efficient numbering method is base 16, or hexadecimal, which interprets a sequence of four binary digits as a singular hexadecimal number. This arrangement implies that just two hexadecimal numbers suffice to represent a single byte[6]. Hexadecimal numbers employ the same symbols as decimal numbers up to 9, subsequently using the characters A through F to denote decimal values from ten to fifteen. For example, the hexadecimal number 10_{16} is equivalent to the decimal number 16_{10}.

Table 1- 1 Binary, Decimal and Hexadecimal equivalents

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A

[6] *A single hexadecimal number is sometimes referred to as a nibble.*

1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

1.3.2 Converting Binary to Decimal

Each binary[7] digit can be converted to decimal by multiplying its value by two raised to an index where the index corresponds to the bit's position.

The binary number 110101_2 then, can be converted to decimal using the following steps.

$1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 =$

$32 + 16 + 0 + 4 + 0 + 1$

$= 53_{10}$

1.3.2.1 General Rule for Base Conversion

Any number *n* in binary can be written as:

n = 10 × quotient + remainder

1.3.2.2 Binary Long Division

Example: n = 101101

[7] *Note these steps use pure binary, it is often faster to temporarily use decimal numbers as interim steps, for example to find out the largest divisor that divides 1010_2 into binary 10111010_2, convert the numbers to decimal to get 10_{10} and 186_{10}, so it is easy to see that 18_{10} is the largest number when multiplied by 10_{10} that will divide into 186_{10}. Converting 18_{10}, back to binary gives 10010_2. Checking 100010_2 x 1010_2 = 10110100_2 which divides into 10111010_2*

Numerator ÷ Denominator = Quotient + Remainder
Numerator —> the number being divided
Denominator —> the number you divide by
Quotient —> the result of the division
Remainder —> what's left over
1 0 1 1 0 1 = 10 X Quotient + Remainder

so divide by the target base which is 10

1 0 1 1 0 1 ÷ 1 0 1 0 (10 decimal in binary)

Quotent >				0	0	0	1	0	0	
1	0	1	0	1	0	1	1	0	1	
				1	0	1	0			subtract
				0	0	0	1			
				0	0	0	1	0		Bring down the next digit from the numerator
Remain >				0	0	0	1	0	1	Doesn't divide so bring down the next digit and place 0 in the next quotient position
										Still does not divide, place 0 in the next quotient position and
										this is the remainder as there are no more numerator digits

This gives a quotient of 4 with a remainder of 5
Verifies n = Base x Quotient plus the Remainder

1.3.2.3 Repeated division method (algorithmic)

- Divide by target base – Here base = 10
- With repeated division, the remainders are the decimal digits.
- The decimal numbers appear in reverse order, with the least significant appearing first.

Example

Convert: 1111001_2 to decimal by dividing by 1010_2 (10_{10})

Repeatedly divide by 1010_2; each remainder is one decimal digit (in binary).

Divide:

$1111001_2 \div 1010_2$

Using trial and error - test multiples of 1010_2:

Attempt $1010_2 \times 1011_2 = 1101110_2$

This divides into 1111001_2 so try next number up -

$1010_2 \times 1100_2 = 1111000_2$, this also divides so try next number

$1010_2 \times 1101_2 = 100000010_2$, too big so 1100_2 is the quotient

Now subtract (the product of the base by the largest divisor) from the number that is to be converted to get the quotient.

1111001

- 1111000

0000001

Quotient = 1100_2 (It divided 1100 times)

Remainder = 1_2 1_{10}

Now divide the quotient by the base 1100 ÷ 1010_2

1100

-1010

0010

Quotient=1 It divided 1 time

Remainder = 10_2 2_{10}

Now divide the quotient by the base 1 ÷ 1010

It did not divide

Quotient = 0

Remainder =1_2 1_{10}

List the remainders in reverse order = 121_{10}

Further example

Convert 10101001 to decimal

1010 x 10000 = 10100000

10101001

-10100000

00001001

Remainder = 9_{10}

10000 (Quotient)

Divide Quotient by the base 10000 ÷ 1010

10000

- 01010

0 1 10

Remainder = 6_{10}

Quotient is 1

Divide quotient by the base 1 ÷ 1010

Divides zero times with 1 left over

Remainder = 1_{10}

Assemble the remainders in reverse order = 169_{10}

Note there are easier ways to perform these calculations, but the steps presented here can be adapted to assembly programming in a more algorithmic method.

1.3.2.4 Converting Decimal to Binary

The following method breaks down a decimal number into powers of two, so to convert the number 843_{10} to its equivalent binary number –

1. First get the highest power of two contained in 843 which is 512 (2^9).
2. Subtract 512 from 843 = 331,
3. The highest power of two contained in 331 is 256 (2^8),
4. Subtract 256 from 331 to get 75,
5. The highest power of two contained in 75 is 64 (2^6),
6. Subtract 64 from 75 to get 11,
7. The highest power of two contained in 11 is 8(2^3),
8. Subtract 8 from11 to get 3,
9. The highest power of two contained in 3 is 2 (2^1),
10. Subtract from 3 to get 1,
11. The highest power of two contained in 1 is 1 (2^0),
12. Subtract 1 from 1 to get 0.

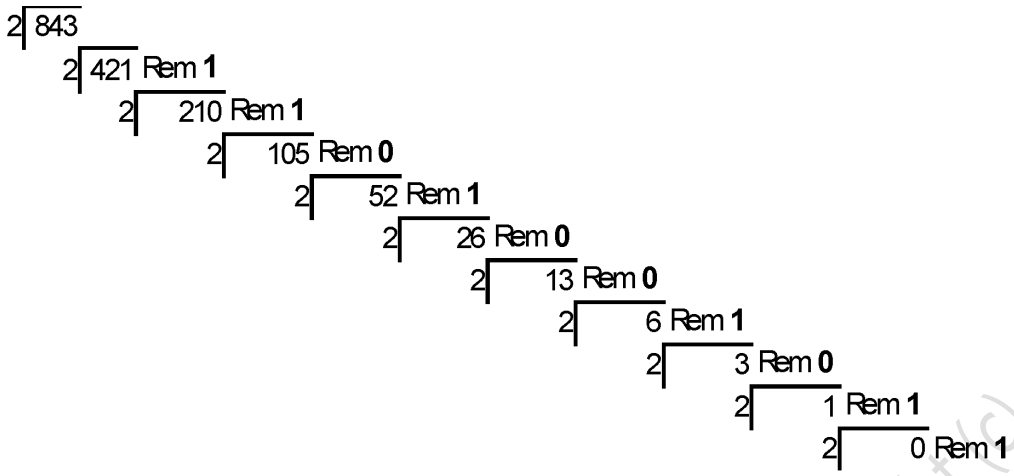
Everywhere that a power of two appears, write its index as the binary value one and where it did not appear write the binary value zero using the positional notation shown in Table 1- 2.

Table 1- 2 Converting decimal to binary

Value	**1**	**1**	**0**	**1**	**0**	**1**
Position	5	4	3	2	1	0
Multiply by	2^5	2^4	2^3	2^2	2^1	2^0

Another way of converting is a repeated division method. Divide the number repeatedly until zero is reached. Take note of the remainders and put the first remainder in the left-most position, then the second remainder into the left-most second position, repeating until all reminders have been recorded.

Figure 1- 1Converting Decimal to binary using repeated division by 2_{10}



Now write down the remainder, starting from the top, to get:

1101001011_2.

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	0	1	0	1	1

1.3.3 Converting Hexadecimal to Decimal

A hex number such as $5B7C_{16}$ can be converted to decimal using a power of sixteen method –

$= 5 \times 16^3, + B \times 16^2, + 7 \times 16^1, + C \times 16^0$

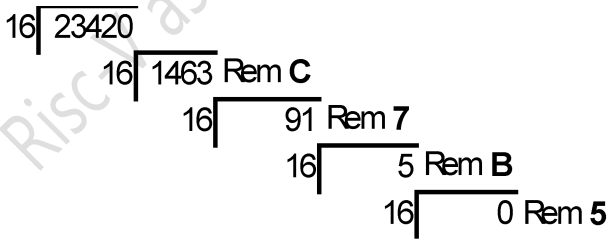
= 20,480 + 2816 + 112 + 12

= 23420

1.3.4 Converting Decimal to Hexadecimal

Take the number as shown, divide repeatedly by 16_{10} until zero is reached. Record the remainders in base 16 format (e.g. for a remainder of 10_{10}, record "A"). Note the remainders and put the last remainder in the left-most position, the second from last remainder in the left-most second position, repeating until all remainders have been recorded.

Figure 1- 2Converting Decimal to binary using repeated division by 16_{10}



Again, printing out the remainders form the bottom gives 5B7C

1.3.5 Binary Fractions

The binary numbers that have been dealt with up to this point are *natural* number equivalents (positive whole numbers). Positional notation is used to show the corresponding power of two index. [8] Fractions can be represented in binary by moving to the left of the 2^0. These values then become 2^{-1}, 2^{-2}, . . .

1.3.6 Converting *a Binary Fraction to Decimal*

1101.01 is equivalent to the base 10 number 13.25 since we have:

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2.}$

1.3.6.1 Converting a Decimal Fraction to Binary

Repeatedly multiply the fractional part by two until it becomes zero, taking note of the value to the left (integer portion) of the decimal point. Accumulate the values of the integer part from top to bottom to get the binary fractional part.

Example 0.625_{10}

0.625 x 2 = 1.25

0.25 x 2 = 0.5

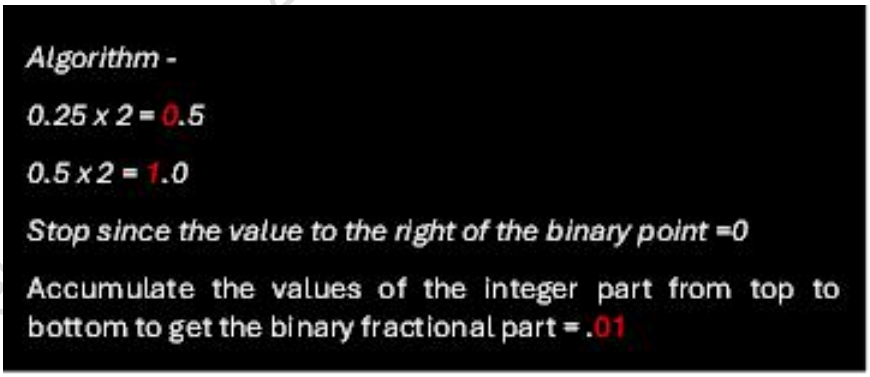
0.5 x 2 = 1.0

Stop since the value to the right of the decimal point =0

Take the integer value from top to bottom = 0.101_2

Consider number 0.3

0.3 x 2 = 0.6
0.6 x 2 = 1.2
0.2 x 2 = 0.4
0.4 x 2 = 0.8
0.8 x 2 = 1.6
0.6 x 2 = 1.2
0.2 x 2 = 0.4
0.4 x 2 = 0.8
0.8 x 2 = 1.6
0.6 x 2 = 1.2



This highlighted value has been met before, so this is a recurring fraction with the pattern 0011 repeating - .0100110011… This means that when evaluating, a halt counter should be added. The logic would be to end when the fractional part = 0 or when the required degree of precision has been reached.

[8] *Recall that negative indices can be resolved by changing the sign of the index and changing the operation from division to multiplication and vice versa so that $1 / 2^{-2}$ becomes $1 \times 2^2 = 4$ and $4 \times 2^2 = 4/2^{-2} = 16$*

1.3.7 One's and Two's complement

An eight-bit byte can represent any one of 256 values ranging from 0 – 255_{10}. This is known as *unsigned* notation. Another representation is to use half of the range as positive integers and the other half as negative; in this case, the range is from +127[9] through -128. This method uses the *most significant bit* to represent the sign and is known as *signed* notation. The number line for an eight-bit signed number is:

-128, -127, …, 0, 1, 2, …, 127

Table 1- 3 Signed number representation.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Sign bit	Magnitude bits						

Interpreting the value of a signed number is straightforward –

The procedure is to add the corresponding powers of two of each bit's place value but leave out the sign bit. The next step is to add in the value of the sign bit. For positive numbers, it makes no difference since the value of the sign bit is zero, but for negative numbers, the value of the sign bit is -128.

Example

- Take the positive binary number 00101100
- Add the magnitude bits together:

$0x2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

$= 32 + 8 + 4 = 44$

- Add in the value of the sign bit (2^7) to get:-

$0 + 44 = 44$

- For the negative number 10011001
- Add the magnitude bits together.

$0x2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 16 + 8 + 1 = 25$

- Add in the value of the sign bit (2^7)to get

$-128 + 25 = -103$

Converting from a signed number to an unsigned number is a simple operation, the procedure is to invert the bits and then add the binary value 1.

[9] *Zero is treated as a positive number here*

So, to convert the positive number 63_{10} to negative 63_{10}.

- Convert the number to an eight-bit binary number -

00111111

- Invert the bits to get -

11000000 (one's complement)

- Add 1 to get –

11000001 (Two's complement)

- Convert back to decimal to get:-

-128+64+1 = 63

1. The first stage of inverting the bits obtains the one's complement; adding the binary digit 1 to the one's complement - obtains the two's complement.

The following table shows an extract of the first few signed numbers.

Table 1- 4 Signed and Unsigned Numbers

Signed Binary Number	Decimal Equivalent
0111 1111	127
0111 1110	126
0111 1101	125
. .	.
0000 0000	0
1111 1111	-1
1111 1110	-2
.. . .	
1000 0010	-126
1000 0001	-127
1000 0000	-128

1.3.8 Addition and subtraction of binary numbers

1.3.8.1 Binary Addition

To add two binary numbers together is straightforward; there are only four outcomes.

0 + 0 = 0

0 + 1 = 1

1+ 0 = 1

1 + 1 =10 (0+ carry)

An example of an unsigned binary addition follows:

Add 0 0 1 0 1 1 0 1 to 0 1 1 1 0 1 0 0

0	**0**	**1**	**0**	**1**	**1**	**0**	**1**
0	1	1	1	0	1	0	0
1	0	1	0	0	0	0	1

Checking by adding the decimal number equivalents together –

45 + 116 = 161

Consider if these numbers being added were in signed notation – here, adding two positive numbers together would result in a negative number since the sign bit of the result = 1. This is an *overflow* condition since the result of 161 is clearly outside the maximum positive number that can be represented in signed eight-bit binary arithmetic. This is something that needs to be checked, and there are conditions built into the processor architecture to detect this kind of situation.

Larger numbers can be dealt with by using two bytes for storage, treating the second byte as having the values 2^8 through 2^{15}. Assemblers and compilers will refer to groups of bytes by designations such as long int, word etc. It is important to check the definitions.

One such definition is:

Table 1- 5 Data type sizes

Unit	Width
Doubleword	64 bits
Word	32 bits
Halfword	16 bits
Byte	8 bits

Of course, it is important to specify signed or unsigned. Again, a definition for an unsigned integer in the programmer's documentation might be referred to as `uint`.

1.3.9 *Binary subtraction*

Binary subtraction can be dealt with using elementary rules for small numbers and then taking into account "borrows" rather than "carrys" but using the two's complement method described on page 1-11 is by far the preferred method for larger numbers.

The steps for binary subtraction are:

1. Obtain the two's complement of the *subtrahend* (the number that will be taken away)
2. Add this to the *minuend* (the number that will be subtracted from).
3. Add the two's complement of the subtrahend to the minuend.
4. If there is a carry after the addition, then drop the carry (the final result is positive)
5. If there is no carry, then compute the two's complement of the result (the final result is negative)

Taking a concrete example of subtracting 00100100 (36_{10}) from 00000010 (2_{10})

- Two's complement of the subtrahend:

1101 1011 +1 = 1101 1100

- Add to the minuend:

0	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**Minuend**
1	1	0	1	1	1	0	0	Two's complement of subtrahend
1	1	0	1	1	1	1	0	

(Carry = 0)

Two's complement of the result is

00100001+1 = 00100010

Result is negative since the carry was false = -34

Another example -

- Subtract 45_{10} from 120_{10}
- Convert numbers to eight-bit binary

45_{10}= 0010 1101_2

120_{10} = 0111 1000_2

- Two's complement of 00101101

1101 0011

- Add to 0111 1000

0	**1**	**1**	**1**	**1**	**0**	**0**	**0**
1	1	0	1	0	0	1	1
0	1	0	0	1	0	1	1

(carry = 1)

The result is positive since carry was zero, 01001011 = 75_{10}

1.3.10 *Binary multiplication*

The rules for multiplication of two bits are

0 x 0 = 0
0 x 1 = 0
1 x 0 =0
1 x 1 =1

Note anything multiplied by zero is of course zero.

Example multiply binary 10 (2_{10}) by 11 (3_{10})

	1	**0**	
	1	1	x
	1	0	
1	0		
1	1	0	

= 6_{10}

Note this is the same as decimal multiplication, where we multiply by each of the digits and then add these results together.

1.3.11 *Binary division*

The rules for division of two bits are as follows (recall that division by zero is invalid)

- 0 / 0 invalid
- 0 / 1 = 0
- 1 / 0 invalid
- 1 / 1 =1

Division example

Divide 1 1011 (Dividend) by 00111 (Divisor)

Using long division -

```
Divide 11011        by      111
              0  0  0  1  1
        111 | 1  1  0  1  1  Bring down
Subtract      1  1  1  ↓     the 1
              1  1  0  1
Subtract         1  1  1
                 1  1  0  ←  Remainder (since it is
                            too small to be divided by 111)
Check    by converting to base 10   27/7    = 3 with remainder 6
Dividend      27
Divisor        7
Quotient       3
Remainder      6
```

1.3.12Shift/ Rotate instructions to perform multiplication and division operations

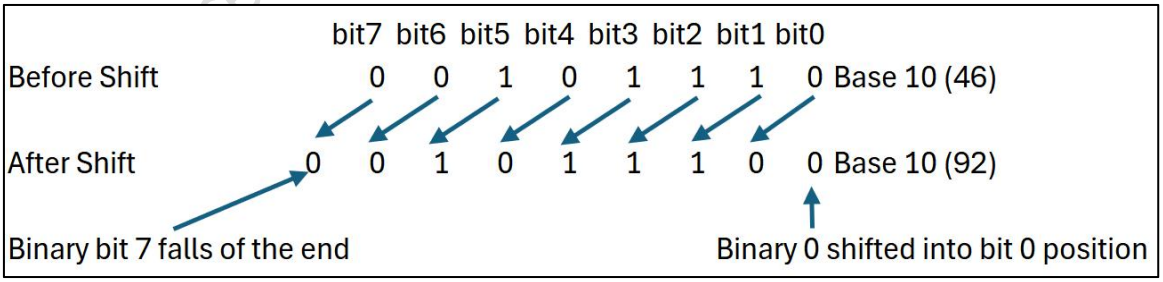
Consider an eight-bit byte 00101110 which has the decimal equivalent of 46. Next, take each bit of the byte and shift them over one place to the left, filling in the now vacant bit 0 with the padded value 0 as shown below. Bit 7 has nowhere to go since it has no bit 8 position to occupy. The newly vacated bit 0 position is filled with a binary zero.

By shifting all the bits to the left, the original number has been *multiplied by two* since the bit 0 value of 2^0 has been moved to the 2^1 position, bit 1's value of 2^1 has been moved to 2^2, etc.

Note that if the original bit 7 had a value of 1 then it would have been lost, giving an incorrect result. This is a condition that *must* be checked for by the programmer and this will be covered in a later section.

Division by two is accomplished by shifting the bit values to the right.

Figure 1- 3 Using shift operations to multiply and divide by two



bit 0 → bit 1→ bit 2 → bit 3 → bit 4 → bit 5 → bit 6 → bit 7 → bit 0, . . .

For simplicity, the registers shown are byte-wide. In reality, the width is more often 32 or 64 bits.

Other rotates are possible, where the shifted-out bit feeds back to the input, giving a circular action.

Bit0 →Bit1 →Bit2 →Bit3 →Bit4 →Bit5 →Bit6 →Bit7 →Bit0 →Bit1…

1.3.13 Binary Coded Decimal (BCD)

Binary Coded Decimal represents decimal numbers in groups of bits, the encoding is normally done in four-bit nibbles. Each bit represents a power of two weight (2^3, 2^2, 2^1, 2^0, or 8,4,2,1). Since four bits can represent 16 distinct numbers, and there are only ten decimal digits, wastage occurs with this method. An alternative known as *packed BCD* may be used, but it is less common.

1.3.13.1 Converting Binary Coded Decimal to Decimal

BCD is similar to hexadecimal except that hex characters a through are illegal. A binary grouping of BCD characters could look like:

1001 0111 1000. Each group of 4 bits (nibbles) are read off as follows –

- 1001 = 9
- 0111 = 7
- 1000 =8

This corresponds to the decimal number 978.

1.3.13.2 BCD addition

Adding is straightforward; however, if the addition of two nibbles results in a value greater than 9 (1010, 1011, 1100,1101, 1110, 1111) then it is an invalid decimal number. The resolution is to add 6 (0110) which will bring it back to a valid number. The carry will be added to the next nibble.

Addition examples –

Step 1.

14 + 22 = 36 = 0011 0110

Verify by binary addition

0001 0100 (14)

0010 0010 (22) +

0011 0110 (36)

Step 2.

20 +20 = 40 = 0100 0000

0010 0000 (20)

0010 0000 (20) +

0100 0000 (40)

Step 3.

26+25 = 51 = 0101 0001

0010 0110 (26)

0010 0101 (25)+

0100 1011 Least significant nibble is greater than 9 so add 6

0000 0110 + (6)

01010001 (51)

Step 4.

121 + 157 = 278 = 0010 0111 1000

0001 0010 0001 (121)

0001 0101 0111 (157)+

0010 0111 1000 (278)

Step 5.

199 + 933 = 1132 = 0001 0001 0011 0010

0001 1001 1001(199)

1001 0011 0011 (933)+

1010 1100 1100 (Two nibbles invalid add 0110 0110

0000 0110 0110 +

1011 0011 0010 Now, the most significant nibble is invalid so add 6 to it

0110 0000 0000 +

0001 0001 0011 0010 (1132) Brings in a fourth nibble!

1.3.13.3 Conversion from Hex/Pure Binary to BCD

One way of converting a hex number to BCD is to convert the hex number to decimal and then to BCD. An alternative is to use the double-dabble method.

1.3.13.4 Double-Dabble

The double-dabble algorithm is fairly simple to implement; it consists of a series of shift[10] operations and additions.

[10] *Shift/Rotate operations are discussed on page 16.*

Note that an n-digit hex number can translate into more than n decimal digits (8516 = 13310, FFF16 = 409510).

The method sets up a store to hold n binary digits and partitions to hold the decimal powers of two – units, tens, hundreds, thousands, … The partitions are cleared to hold all zeros, and then the binary digits are shifted in one bit at a time. Adjustments (addition of decimal 3) are made to the partition values dependent on their magnitude (>4). Once all bits have been shifted[11] the algorithm has been completed.

An example:

Consider the binary number 00011011 = hex 1B = decimal 27. The steps to convert from pure binary to BCD are shown in Table 1- 6.

Table 1- 6 Double-Dabble example

Hundreds Partition	Tens Partition	Units Partition	Binary Store	Action
0000	0000	0000	00011011	
0000	0000	0000	00110110	Shift left-most bit over to partitions (shift1)
0000	0000	0000	01101100	Shift left-most bit over to partitions (shift2)
0000	0000	0000	11011000	Shift left-most bit over to partitions (shift3)
0000	0000	0001	10110000	Shift left-most bit over to partitions (shift4)
0000	0000	0011	01100000	Shift left-most bit over to partitions (shift5)
0000	0000	0110	11000000	Shift left-most bit over to partitions (shift6)
0000	0000	1001	11000000	Add 3 to units, since unit is 5 or greater
0000	0001	0011	10000000	Shift left-most bit over to partitions (shift7)
0000	0010	0111	00000000	Shift left-most bit over to partitions (shift8)

Reading off the tens and unit columns gives the value 27_{10}.

Note 3 is added rather than 6 since the shift left operation multiplies by two!

A more complex 12-bit example is shown in Table 1- 7.

[11] *The number of shifts is equal to the number of binary digits*

Table 1- 7 Three digit double dabble example

Double Dabble Three digit Hex (200) number
12 binary digits so 12 shifts are required

Hundreds	Tens	Units		Binary		Initial State
0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	0 0 0 0	Initial State
0 0 0 0	0 0 0 0	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	Shift #1
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0	Shift #2
0 0 0 0	0 0 0 0	0 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	Shift #3
0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #4
0 0 0 0	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #5
0 0 0 0	0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #6
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 0 0	0 0 0 0	1 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 0	0 0 0 1	0 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #7
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 0 0	0 0 0 1	1 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 0	0 0 1 1	0 0 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #8
0 0 0 0	0 1 1 0	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #9
0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to tens
0 0 0 0	1 0 0 1	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 1	0 0 1 0	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #10
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 0 1	0 0 1 0	1 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 1 0	1 1 0 1	0 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #11
0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to Tens
0 0 1 0	1 0 0 0	0 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 1 0	1 0 0 0	1 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 1 0 1	0 0 0 1	0 0 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #12
5	1	2	200 hex = 001000000000 binary = 512 decimal			

1.3.14Floating-Point

Integers, such as 107 or 456, are defined as whole, complete, and exact numbers. A simple storage unit, like a register, places a limit on the magnitude that can be represented. In contrast, floating-point representation enables the depiction of exceptionally large or small numbers, although this is achieved at the expense of precision. Consequently, a floating-point number may merely be an approximation, leading to *rounding* of digits. A floating-point number fundamentally consists of two primary components: the *significand* (or *mantissa)* and the *exponent*. Additionally, provision is made for a sign bit. The structure is the *significand* multiplied by the *base* elevated to a *power*. For example, 3,450,000 can be expressed as $345 \times 10^{\wedge 4}$, where 345 is the significand, ten is the base, and four is the exponent.

A standard known as *IEEE 754* (https://standards.ieee.org/ieee/754/6210/) specifies the rules for floating-point arithmetic. This particular standard details both Single and Double precision floating-point formats[12], which are presented in Table 1-8. It also makes provisions for incorporating special values like Not-a-Number[13] (NaNs) and ±Infinity.

A 32-bit single-precision floating-point binary number within IEEE 754 is defined as:

Sign Bit (1 bit)	Exponent (8 bits)	Significand (23 bits)

A 64-bit double-precision floating-point binary number within IEEE 754 is defined as:

Exponent (11 bits)	Significand (52 bits)

This is summarized in Table 1- 8.

Table 1- 8 Floating-Point formats

Format	Bits	Significand	Unbiased Exponent	Decimal Precision
Single	32	24 [14] (23+1)	8	6-9 digits
Double	64	53 (52+1)	11	15-17 digits

1.3.14.1 Biased exponents

The use of a biased exponent can represent negative exponents. For single precision the values range from decimal +127 to -126. The bias is normally given as 2^{n-1}-1 where n is the number of exponent bits, so here we have 2^7-1= 127. The value of the biased exponent is the unbiased exponent minus 127, so that an exponent of 10011011 gives a biased exponent of (128+16+8+2+1) – 127 = 155-127 = 28.

1.3.14.2 Infinity and Not-a-Number representation

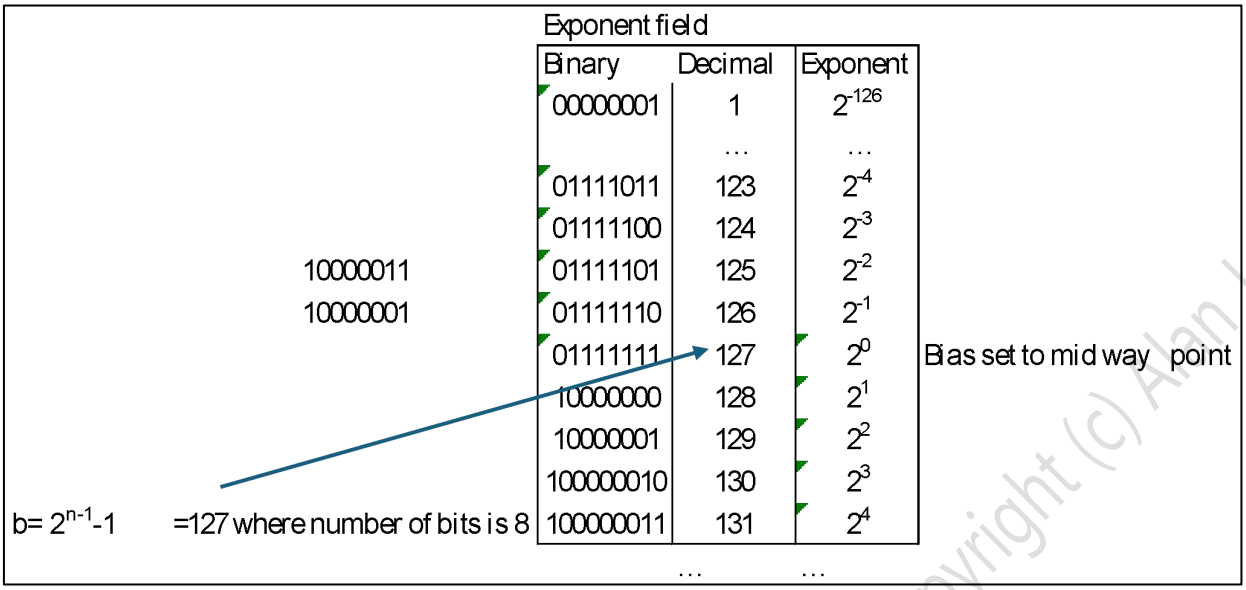
- A biased exponent of all ones and a significand of all zeros (-127) represents infinity. The sign bit differentiates between negative and positive infinity.
- Not-a-number is represented by the biased exponent being equal to all ones (+128) and the significand being non-zero.
- The sign bit is don't care.

[12] *Other formats are defined but they will not be discussed here.*

[13] *This could arise from operations such as divide by zero or the square root of a negative number.*

[14] *There is an implied bit, since the normalized format is always 1.X then there is no need to specify the "1" value to the left of the decimal point.*

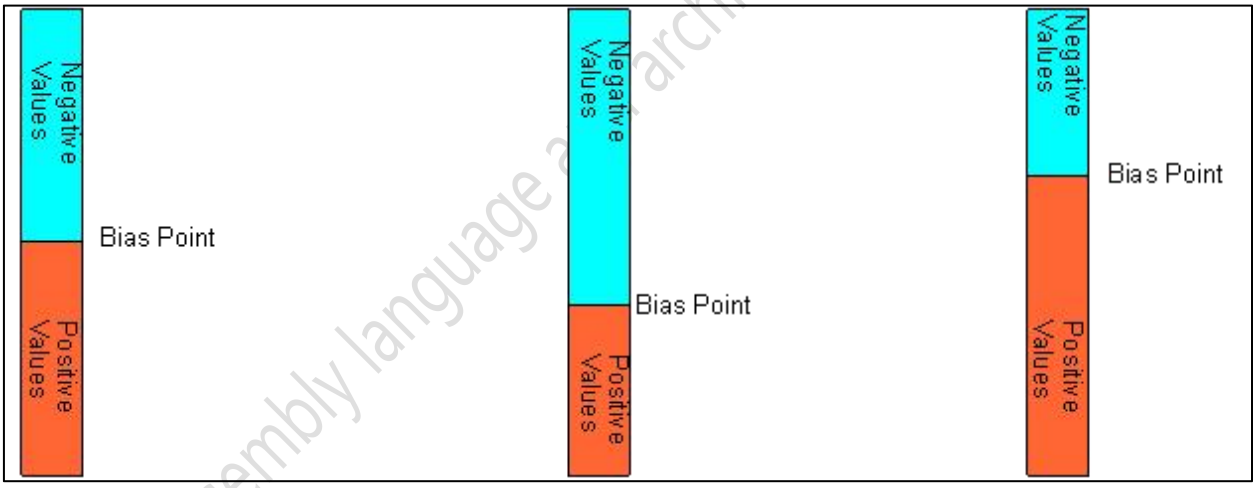
Table 1- 9 BIAS within single precision IEEE 754



1.3.14.3 Understanding bias

The diagram shown in Figure 1- 4 shows how varying the bias affects the ratio of negative to positive numbers. The bias used in the standard gives similar *ranges* of positive and negative exponents.

Figure 1- 4 Interpretation of Bias with floating-point



With double-precision numbers, the bias is 1023 since the unbiased component shown in Table 1- 8 is 11-bits wide.

1.3.14.4 Normalized numbers

A normalized number has the form 1.XXXXX… The steps are to convert the number to binary and then perform shifts to give the desired result. Normalization shifts to the left or right depending on where the decimal point is.

Example 410.625

Steps -

Step 1 Convert to binary

= 110011010.101

Step 2 Perform repeated shift until the desired pattern is reached.

110011010.101 ÷2 (shift right operation)

= 11001101.0101 ÷2
= 1100110.10101 ÷2
= 110011.010101 ÷2
= 11001.1010101 ÷2
= 1100.11010101 ÷2
= 110.011010101 ÷2
= 11.0011010101 ÷2

=1.10011010101

This took a total of 8 shift operations. Add this number to 127 to get 135. Convert to binary to get:

10000111.

From our shifts earlier, we had the value 10011010101, extend this to 23 bits to get 10011010101000000000000 giving the value:

S	Exponent								Significand																		
0	1	0	0	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0

= 410.625

1.3.14.5 Encoding a number to floating-point (single precision)

This example shows how to encode the decimal number minus 13.25 into single precision floating-point.

Step 1

Determine the sign

The sign is negative.

Step 2

Convert to binary.

The part to the left of the decimal point is straightforward as 13 is equal to 1 1 0 1 binary. The fractional part can be computed using the algorithm discussed earlier and shown here. Multiply the fractional part by two until the value is zero, noting the value to the left of the decimal point. Here multiplying .25 by two gives a value of 0 to the left of the decimal point and 5 to the right of the decimal point. Multiply again, and this time there is a zero to the right of the decimal point, so we stop and record the value of 1 to the left of the decimal point. Recording the values to the left of the decimal point from the top down gives a value of 01. Combine to get and absolute value of 1101.01.

Step 3

Normalize

Next the value is normalized by shifting three places to get 1.1 0 1 0 1. The number then is 1.10101 by 2^3 (because it was moved over 3_{10} or 11_2 places)

Step 4

Bias and generate the new exponent

The bias for single precision is 1 1 1 1 1 1 1 plus the stored exponent of 11 giving 1 0 0 0 0 0 1 0 for the exponent value.

Step 5

Build the significand

Drop the leading one from the normalized value (because it is implicit in IEEE-754) to get a significand of 1 0 1 0 1 and pad the remainder of the field to get 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ,

Step 6

Assemble the fields

Construct the full value with the sign bit, exponent and the significand to get a full 32 bit number of 11000001010101000000000000000000.

1.3.14.6 Encoding a number to floating-point (double precision)

Most of the steps are similar to the previous example, the differences are in the bit length of the exponent and significand fields.

- Example: - 13.25

Step1

Determine Sign

Negative

Step 2

Convert to Binary

Integer part: 13

13 in binary = 1101_2

Fractional part: 0.25

0.25 × 2 = .01

Combine to get value = 1101.01

Step 3

Normalize

Move the binary point over three places so the number becomes: $1.10101_2 \times 2^3$

Significand: 1.10101

Exponent: 3

Step 4

Bias and generate the new exponent

Bias = 1023

Stored exponent = 3 + 1023 = **1026**

1026 in binary = **10000000010**

Step 5

Build the significand

Drop the leading 1 (implicit in normalized form).

Normalized significand:

Double precision: 52 bits

1.10101 → 1010100000000000000000000000000000000000000000000000

Step 6

Assemble the fields

Sign: 1

Exponent: 10000000010

Significand: 1010100000000000000000000000000000000000000000000000

Full 64- bit pattern: 1100000000101010100000000000000000000000000000000000000000000000

1.3.14.7 Decoding a single-precision floating-point number

This example will show how to decode the single-precision floating-point number 0x C40DC000

Step 1

Convert to binary

= 1100 0100 0000 1101 1100 0000 0000 0000

Step2

Determine the sign

Sign is negative.

Step 3

Breakdown the fields

1 10001000 00011011100000000000000

Number is negative

Exponent = 136 = (136-127) = 9

Step 4

Build significand and convert to base 10

Implicit bit 1. 00011011100000000000000

Convert .000110111 portion of significand (other bits are all zeros)

$= 0 x2^{-1} + 0 x 2^{-2} + 0 x 2^{-3} + 1 x 2^{-4} + 1 x 2^{-5} + 0 X 2^{-6} + 1 x 2^{-7} + 1 x 2^{-8} + 1 x 2^{-9}$

=1/16 + 1/32 + 1/128 + 1/256 + 1/512 = 55/512 = .107421875

=1. 107421875

Step 5

Compute magnitude

Exponent is 2^9 = 512

512 x 1.107421875 = 567

With sign bit = -567.

1.3.14.8 Addition of floating-point numbers

Addition is reasonably straightforward; the main concern is when the exponent differs. To equalize the exponents, take the lower number and shift over the binary point the required number of positions. So, if one exponent is 136-*Bias* and the second is 134-*Bias*, the second number needs to be shifted two places.

Example Add two single precision floating-point numbers together.

The numbers are

01000011011111000011001110001000 0

And

01000001110101010101010101010101

Step 1

Determine the exponents.

Number1's exponent is 134, number2's exponent is 131

Shift the significand of the number with the lower exponent over the required number of places, which in this case is 3 since the difference in bias is 134-131=3. We should already be familiar with this type of operation in base 10 where the general case is Significand × base^{exponent} , so a number such as 9000 x 10^3 can be expressed with an exponent of 6 by diminishing the significand's digits (6-3) places to the right to get 9 x 10^6. In the figure, the exponents are first converted to base 10, for clarity; however, this can be done directly in binary since subtracting 10000011 from 10000110 is 011.

Step 2

Equalize the exponents

Add the implicit "1" bit and shift the second number's significand by 3 places to the right.

1.10101010101010101010101 → 0.00110101010101010101010101 (Last 3 bits are pushed out)

Step 3

Add the significands

	1.	1	1	1	1	0	0	0	0	1	1	0	0	1	1	1	0	0	0	1	0	0	0	0
	0.	0	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
1	0.	0	0	1	0	0	1	1	0	0	0	1	0	0	0	1	1	0	1	1	1	0	1	0

Prepend the implicit leading "1" and add number1's significand to the new value of number2's significand

Step 4

Normalize

Since the addition spilled over and generated two digits to the left of the binary point, the result must be shifted one place to the right. This gives

1.00010011000100011011101

The least significant bit with the value of 0 has been shifted out. The shift has caused the exponent to be increased by one (the reason was described earlier 1). The new exponent field is 135 base 10, 10000111 base 2.

Step 5

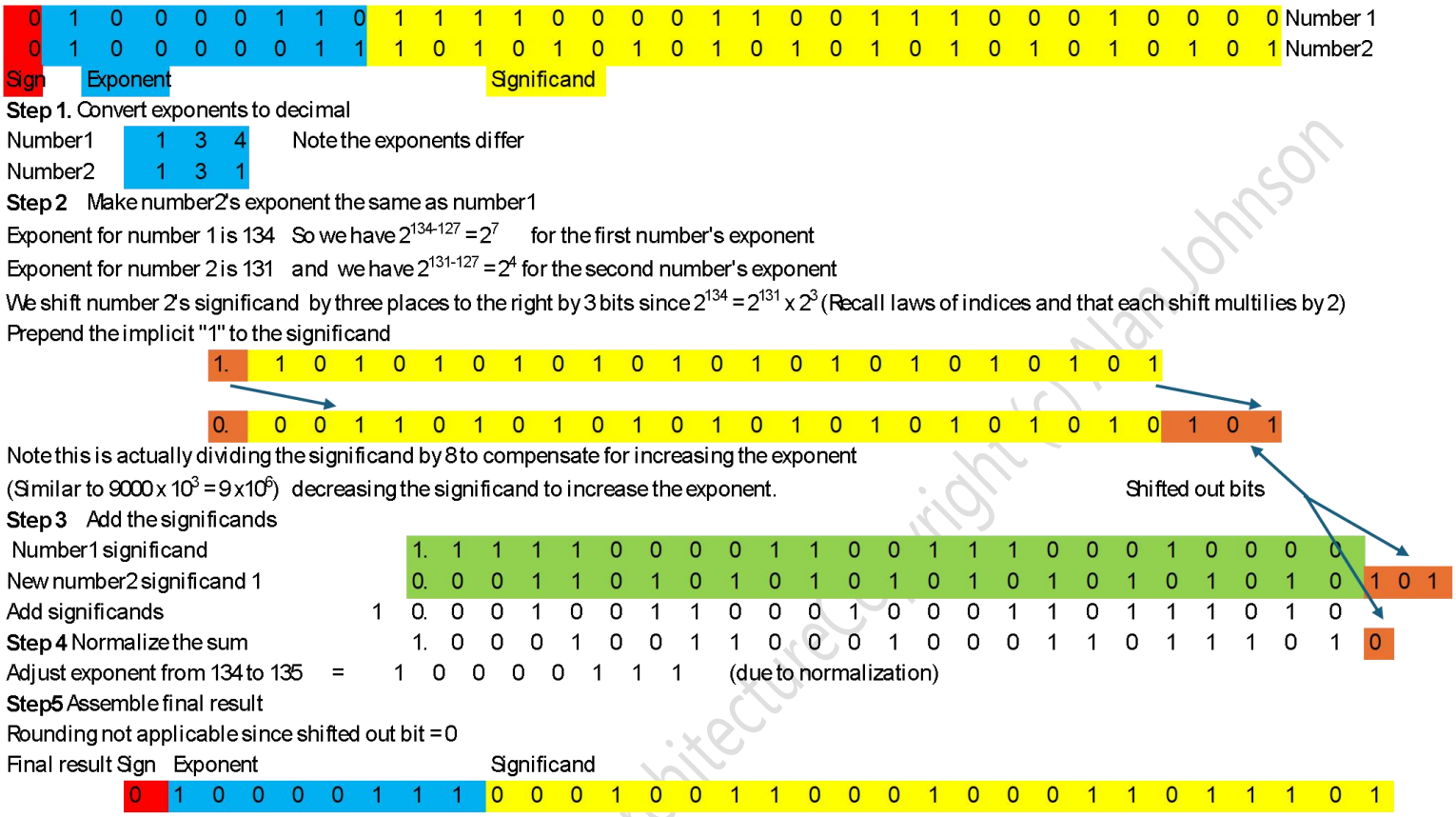
Assemble the fields (dropping the implicit bit)

The final result is Sign bit, exponent, significand =

0 10000111 00010011000100011011101

The steps are shown in Figure 1- 5

Figure 1- 5Addition of two floating-point numbers (Single precision)



1.3.14.9 Further floating-point example

Multiply the decimal numbers 13.5 and 4.25, express the result as a single precision floating-point number. Since we have the two numbers expressed in base 10, it is easier to perform regular base 10 multiplication and then encode the result as a floating-point number. In many cases, it may also be helpful, in examples such as the one shown previously, to convert given floating-point numbers back to base 10 first.

Step 1

Multiply the numbers in base 10 format

13.5 x 4.25 = 57.375

Step 2

Convert to binary

Integer part 57 = 111001

Fractional part .375

.375 x 2 = 0.75

0.75 x 2 = 1.50

0.50 x 2 = 1.00 (Stop)

= 011

= 111001.011

Step 3

Normalize

1.11001011

(shift of 5 places

Step 4

Bias and exponent

Bias = 127 + 5 = 132 = 10000100

Step 5

Build significand

Drop the leading "1" = 11001011

Pad remaining bits 11001011000000000000000

Step 6

Assemble

Sign is positive

0 10000100 11001011000000000000000

In hex = 42658000_{16}

1.4 Logic operations – AND, OR, Exclusive OR, NOT

Logic operations are often used in decision-making, for example –

1. "If I feel hungry AND I have enough money, then I will order food in".
2. "If it is cold OR it is raining, then I will wear a coat to go outside".
3. "I can get a car discount if I pay the total amount in cash OR a I can get a lower interest rate if I take out a loan."

Statement 1 is an AND condition, and the decision to order food holds true if I am hungry AND I have enough money. Both conditions must be true.

Statement 2 is an OR condition, and it states that I will wear a coat if either of these (or both) conditions are true.

Statement 3 is like statement 2 except that it is an either-or situation. Statement 2 applies equally well to both conditions in that it could be cold and also raining (similar to the AND condition). Statement 3 *exclusively* applies to the OR situation and is referred to as Exclusive OR *(XOR*).

These conditions are normally represented by truth tables, such as if condition A is true AND condition B is true, then result C is true. *True* and *false* values can be conveniently mapped to binary values 1 and 0. These are known as *Boolean* variables.

Table 1- 3 Truth table - AND

A	B	C
False (0)	False (0)	False (0)
True (1)	False (0)	False (0)
False (0)	True (1)	False (0)
True (1)	True (1)	True (1)

Table 1-4 Truth table - OR

A	B	C
False (0)	False (0)	False (0)
True (1)	False (0)	True (1)
False (0)	True (1)	True (1)
True (1)	True (1)	True (1)

T able 1- 5 Truth table - XOR

A	B	C
False (0)	False (0)	False (0)
True (1)	False (0)	True (1)
False (0)	True (1)	True (1)
True (1)	True (1)	False (0)

Other logic functions exist, such as NOT which inverts the value, so a binary zero becomes a binary one. Repeating the operation, of course, gets back to the original value. Boolean algebra is a complex topic by itself – which is dealt with in set theory.

For fun, a *simple* encoding can be done with XOR – take the word “Plaintext”, converting this to seven-bit ASCII[15] code becomes –

[15] *See the appendix for a table of ASCII codes*

Table 1- 6 Simple example of encoding text using XOR

Text string	ASCII code (decimal)	ASCII code (binary)	Apply XOR function with 10101010	Resultant ASCII code letter
P	80	1010000	1111010	z
l	108	1101100	1000110	.
a	97	1100001	1001011	K
i	105	1101001	1000011	C
n	110	1101110	1000100	D
t	116	1110100	1011110	^
e	101	1100101	1001111	O
x	120	1111000	1010010	4
t	116	1110100	1011110	^

So, the encoded string “Plaintext” becomes “z.KCD^O4^”.

Of course, this is easily cracked and decoded!

The following rules show the resulting bitwise values:

- X AND 0 = 0
- X AND 1 = X
- X OR 0 = X
- X OR 1 = 1

Now that the foundation is in place it is time to move from generic concepts to programming on a specific architecture!

1.5 Summary

- Introduction to Assembly Language
- Number Systems
- Shift Operations
- Logic and Truth Tables

1.6 Exercises for chapter1

1. Convert 11.110 to base 10
2. Divide 10111101 by 111 using manual long division.
3. Convert 0x1fd to BCD
4. Convert 35.65 to single-precision floating-point according to IEEE 75.
5. Write a pseudocode program to convert lowercase ASCII characters a-z to uppercase ASCII character A-Z.
6. Convert the signed binary byte to base10
7. Convert the octal number 341 to base 16
8. What are mnemonics?
9. Describe the advantages of a high-level language over assembly language.

2 Getting Started

Overview of the chapter

Chapter 2 introduces the RISC-V architecture and the essential tools required to start programming with RISC-V. It moves from theory to practical steps, providing context, tools, and setup guidance for working in RISC-V assembly.

2.1 Origins of RISC-V

The design of RISC-V uses a Reduced Instruction Set Computer (RISC) architecture. RISC-V originated in 2010 as a project at the University of California, Berkeley. The suffix "V" indicates that it is the fifth generation of the RISC architecture. RISC has the advantage of a simpler design with lower power consumption, which makes it ideal for use in embedded systems. RISC-V is now under the stewardship of RISC-V International, based in Switzerland.

A distinguishing feature is that it is open and royalty-free.

2.2 Architecture

Implementations use a naming convention to denote which Instruction Set Architectures (ISAs) are available within a specific implementation. An example is *RV64I* or *RV32E* which stands for RISC-V with a 64-bit *integer* instruction set and RISC-V with a 32-bit reduced *integer* set, respectively[16]. The integer and reduced integer designations form the *Base Integer ISA*. This is mandatory for implementations. Optional extensions are defined as:

- M for integer multiplication and division.
- A for Atomic extensions.
- F and D for single and double-precision floating-point. Here the designation RV64IM would mean 64-bit with Integer and integer multiplication/division support.
- C for compressed instructions.
- E for embedded.
- G for general covers MAFD.
- There is also the ability to support non-standard extensions.

To show RISC-V instruction set support under Linux, the command `cat /proc/cpuinfo` can be used –

```
cat /proc/cpuinfo

processor       : 0

hart            : 1

isa             : rv64imafdc_zicntr_zicsr_zifencei_zihpm_zba_zbb

mmu             : sv39
```

[16] *RV128 definitions also exist but will not be discussed here.*

```
uarch        : sifive,u74-mc
mvendorid    : 0x489
marchid              : 0x8000000000000007
mimpid       : 0x4210427
. . .
processor    : 3
hart         : 4
isa          : rv64imafdc_zicntr_zicsr_zifencei_zihpm_zba_zbb
mmu          : sv39
uarch        : sifive,u74-mc
mvendorid    : 0x489
marchid              : 0x8000000000000007
mimpid       : 0x4210427
```

This system is identified by the string `rv64imafdc` has 4[17] CPU cores and supports (**I**)nteger, (**M**)ultiplication/division, (**A**)tomic and (**F**)single and (**D**)ouble precision floating-point, with the ability to handle the smaller code size of (**C**)ompressed instructions. A designation of G represents IMAFD. The architecture shown is 64-bit. The four processors (0-3) are associated with four *harts* (1-4). A hart is a *hardware thread*[18] that can execute its own set of instructions independently of the others. Usually there is a one-to-one correspondence between harts and processors.

The next output is taken from a Banana Pi BPI-F3 system showing eight processors with rv64imafdc**v** support.

Note the inclusion of the V-extension, which is the vector ISA extension. Vector support is an additional bonus, as few systems today support vector operations.

```
$ cat /proc/cpuinfo
processor       : 0
hart            : 0
model name      : Spacemit(R) X60
isa                                                                    :
rv64imafdcv_zicbom_zicboz_zicntr_zicond_zicsr_zifencei_zihintpause_zihpm_zfh_zfhmin_zca_zcd_zba_zbb
_zbc_zbs_zkt_zve32f_zve32x_zve64d_zve64f_zve64x_zvfh_zvfhmin_zvkt_sscofpmf_sstc_svinval_svnapot_svp
bmt
mmu             : sv39
uarch           : spacemit,x60
mvendorid       : 0x710
```

[17] *Only the first and last CPU cores are shown in the output.*

18 A hardware thread is distinct from a software thread. Software threads are multiplexed tasks, controlled by techniques such as time-slicing giving the illusion of separate tasks, whereas are hardware threads are true independent execution units.

```
marchid         : 0x8000000058000001

mimpid          : 0x1000000049772200

. . .

processor       : 7

hart            : 7

model name      : Spacemit(R) X60

isa                                                                                                 :
rv64imafdcv_zicbom_zicboz_zicntr_zicond_zicsr_zifencei_zihintpause_zihpm_zfh_zfhmin_zca_zcd_zba_zbb
_zbc_zbs_zkt_zve32f_zve32x_zve64d_zve64f_zve64x_zvfh_zvfhmin_zvkt_sscofpmf_sstc_svinval_svnapot_svp
bmt

mmu             : sv39

uarch           : spacemit,x60

mvendorid       : 0x710

marchid         : 0x8000000058000001

mimpid          : 0x1000000049772200
```

Currently, there are four (separate) *base* ISAs with discussion on a 128-bit (128I) implementation. A summary is shown in Table 2- 1.

Table 2- 1 Base integer instruction set variants

Name	Address Space/Register Width
RV32I	32-bit
RV64I	64-bit
RV32E	32-bit
RV64E	64-bit

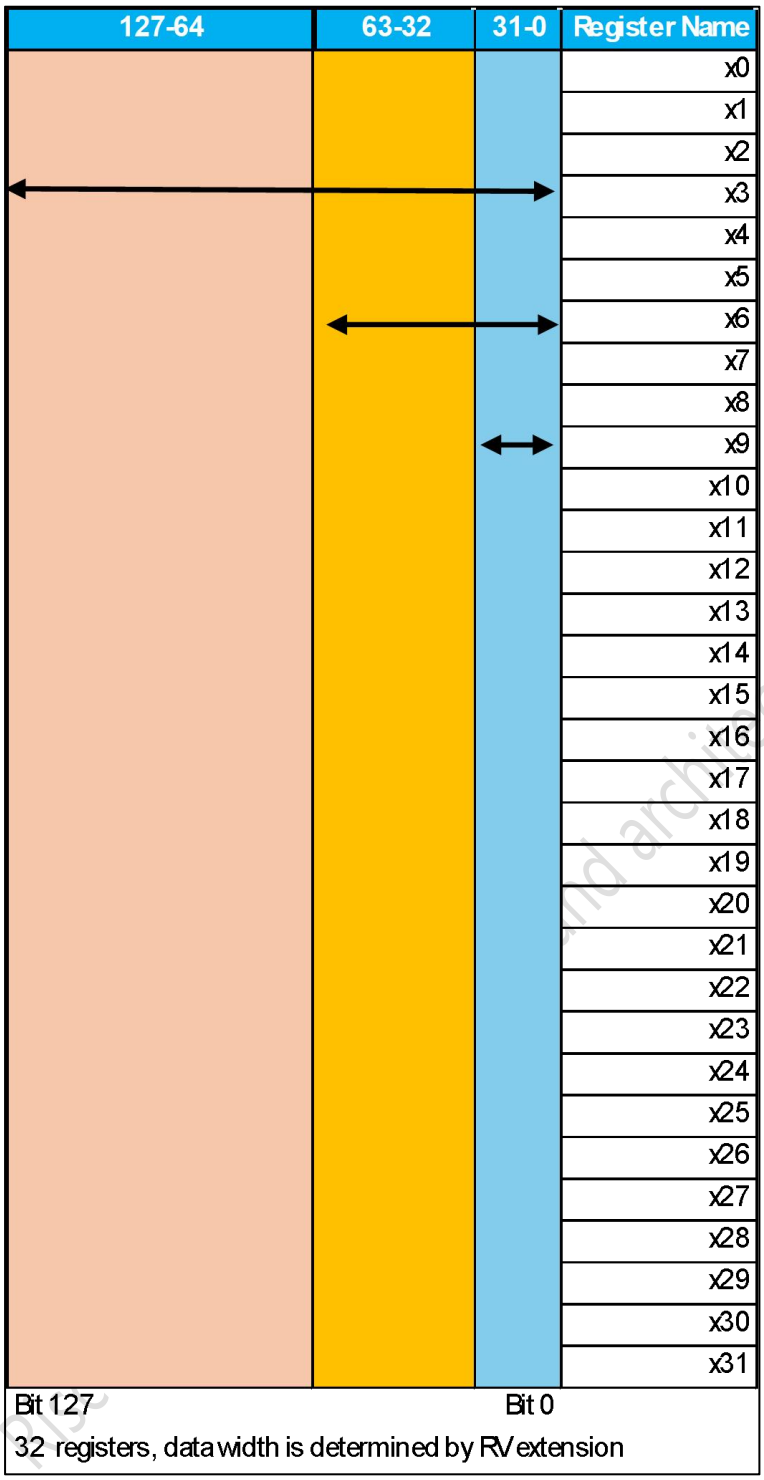
2.2.1 RISC-V Registers

Registers are locations that store values, they are similar to variables in high-level languages.

The primary way of interfacing with the RISC-V system is via the register set. Generically the registers may be referred to as Rd (destination register), Rs1 (first source register), Rs2 (second source register). Registers are denoted by their Application Binary Interface (ABI) name to make it more convenient to the coder. This is similar to high level languages where variables are given meaningful descriptive names.

2.2.1.1 Register Set

Figure 2- 1 RISC-V register layout



There are 32[19] unprivileged *integer* X registers whose width is determined by the instruction set, which is either 32,64 or 128 bits, as shown in Figure 2- 1. The registers have aliased names which reflect their usage such as X1 = ra which holds the return address or X0 = zero which is a read-only register returning the value 0. The aliased names are referred to as the ABI (Application Binary Interface) register name. Even though the registers are general-purpose, the aliased name function should be respected. For example, the saved and temporary registers are used for functions where the coder knows when to save registers prior to making the call and when they do not need to.

When the programmer calls a routine, it is termed the *calling* routine, and the routine that is being called is the *callee* routine. The temporary registers (t0-t6) are saved by the caller, and the saved registers (s0 – s11) are saved by the *callee*. This responsibility is shown in Table 2- 2.

It is only necessary to save the registers that are involved in the routines. So, if the caller was not using register t1 then it would not be necessary to save it prior to involving the call.

There are also 32 floating-point registers accessible to the programmer, which will be discussed at a later point in the book.

The program counter (PC) keeps track of program execution and is not used as a general-purpose register. *XLEN* refers to the data width, which is either 32, 64 or 128 bits. Register functions will be covered in more detail as the book progresses.

19 RV32E and RV64E have 16 registers. The non-contiguous layout is for consistency between register sets

Table 2- 2Caller/Callee Responsibility for X registers

Register Name	ABI Name	Saver responsibility	Register Name	ABI Name	Saver responsibility
x0	zero	N/A	x16	a6	Caller
x1	ra	Caller	x17	a7	Caller
x2	sp	Callee	x18	s2	Callee
x3	gp	N/A	x19	s3	Callee
x4	tp	N/A	x20	s4	Callee
x5	t0	Caller	x21	s5	Callee
x6	t1	Caller	x22	s6	Callee
x7	t2	Caller	x23	s7	Callee
x8	s0/fp	Callee	x24	s8	Callee
x9	s1	Callee	x25	s9	Callee
x10	a0	Caller	x26	s10	Callee
x11	a1	Caller	x27	s11	Callee
x12	a2	Caller	x28	t3	Caller
x13	a3	Caller	x29	t4	Caller
x14	a4	Caller	x30	t5	Caller

This table shows that the temporary (t0-t7) and the argument registers (a0-a7) should be saved by the caller, and the saved registers (s0-s11) by the callee.

2.2.1.2 RV32I Base Instruction Set

The instructions are 32 bits wide; the general format is to include common fields such as:

Field Name	Role
rd	Destination Register
rs1	Source Register 1
rs2	Source Register 2
opcode	Operation Code

func	3 bit (funct3) and 7 bit (funct7) define a particular operation
imm	Constant value such as 0x5F

2.2.1.3 Base Instruction Formats

There are 4 Base Instruction Formats known as:

- I-type
- R-type
- S-type
- U-type

In addition, there are two variants of the I-type instruction known as *B-type* and *J-type*. These instructions use conditional and unconditional branches, respectively, to alter program flow. The immediate fields are used to encode the branch destination.

Conditional branches depend on whether certain program events have occurred; an example could be a *decrementing counter*, where a branch in the program logic occurs only if the counter has reached the value zero.

An unconditional branch happens regardless of conditions. An example might be a jump to an *interrupt handler service routine*[20] if a critical or non-critical event was encountered.

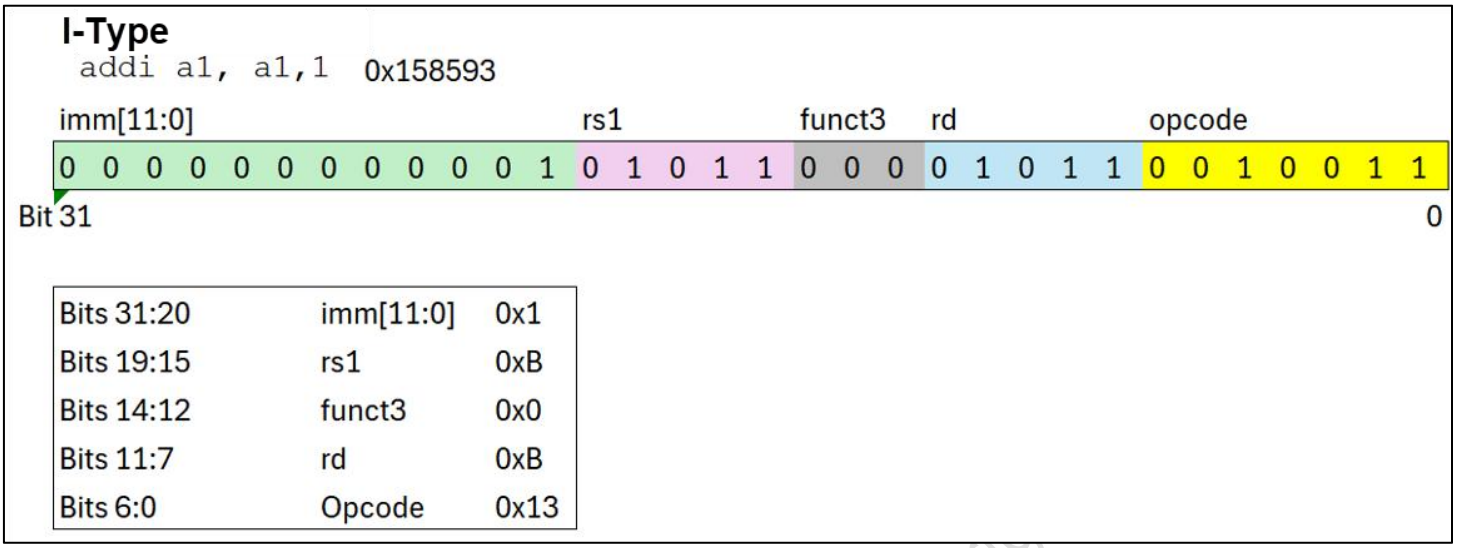
2.2.1.3.1 I-type instruction

Most of the fields occupy the same bit positions across instructions. An example is the instruction `addi a1,a1,1`, which disassembles to 0x00158593. The breakdown of the fields for this instruction is shown in Table 2- 3.

The `ADDI` instruction format is termed *I-type* for immediate. The instruction adds the contents of register a1 plus a constant of 1 to register a1, so the effect is to increase the value of a1 by 1. The a1 register in both the rd and rs1 fields corresponds to the value 0xb which is the 11th X register; so although the programmer uses the more friendly register name, the machine code uses the x register number.

[20] *An interrupt is normally encountered in system level programming and could be used to process an attempt to access kernel memory or a user level action such as a mouse click.*

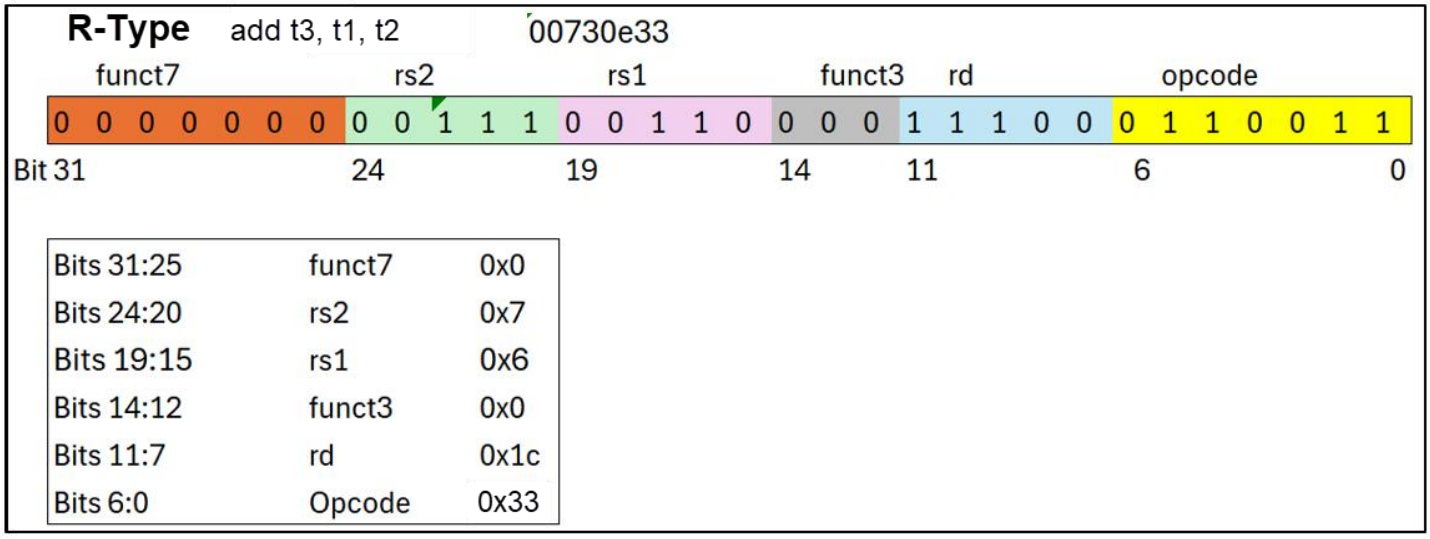
Table 2- 3 Bit fields of the addi I-type instruction



2.2.1.3.2 R-type instruction

The next instruction `add t3, t1, t2` is an *R-Type* instruction as it uses the registers for both operands. Register t2 is added to register t1 and the result is placed in register t3. Disassembly produces the machine code 0x00730e33. The field breakdown is shown in Table 2- 4

Table 2- 4 Bit fields of the add R-Type instruction

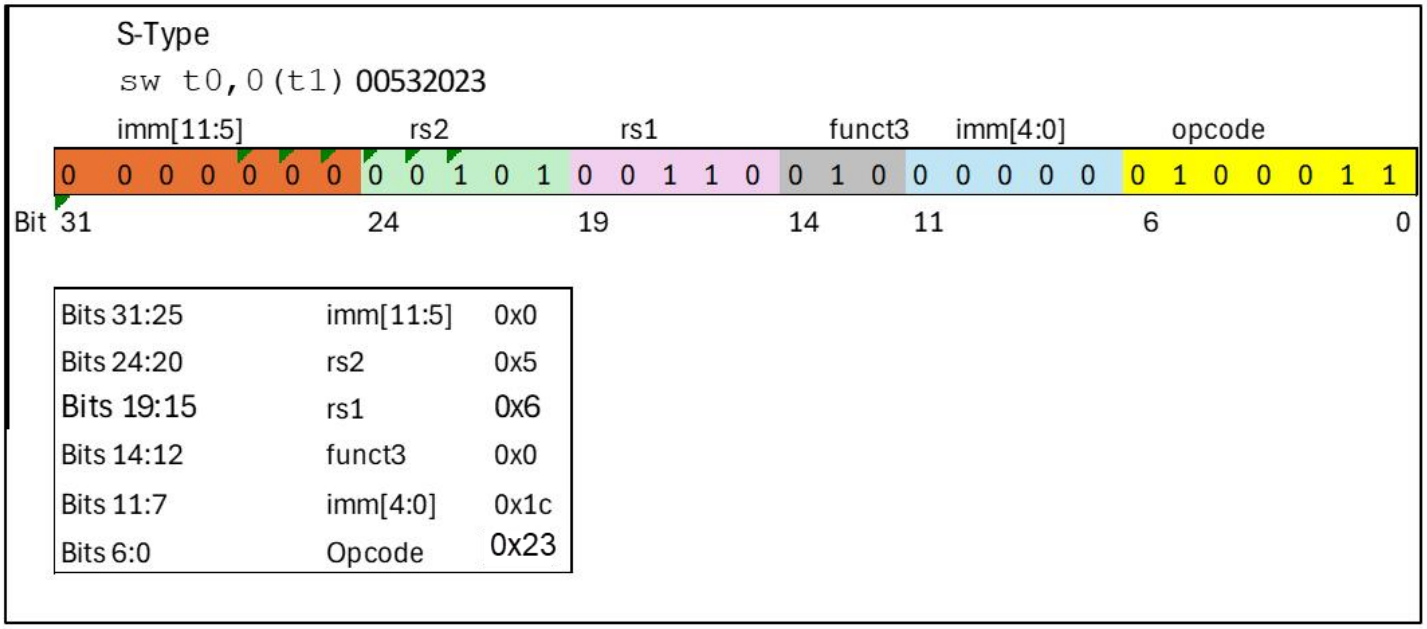


2.2.1.3.3 S-type instruction

The Store Word instruction (`sw`) uses a register and an offset to calculate the destination address. This is an *S-type* instruction. No destination register is involved since the destination is memory. The instruction stores the 32-bits held in register t0 into the memory location pointed to by register t1 plus an immediate offset of 0. The format of the S-Type instruction is shown in Table 2- 5. Note that the immediate data is broken down into two separate fields with the lower 5 bits replacing the unused (in this type of instruction) destination (rd) field.

Disassembly produces the machine code 0x00532023.

Table 2- 5 Bit fields of the sw S-Type instruction



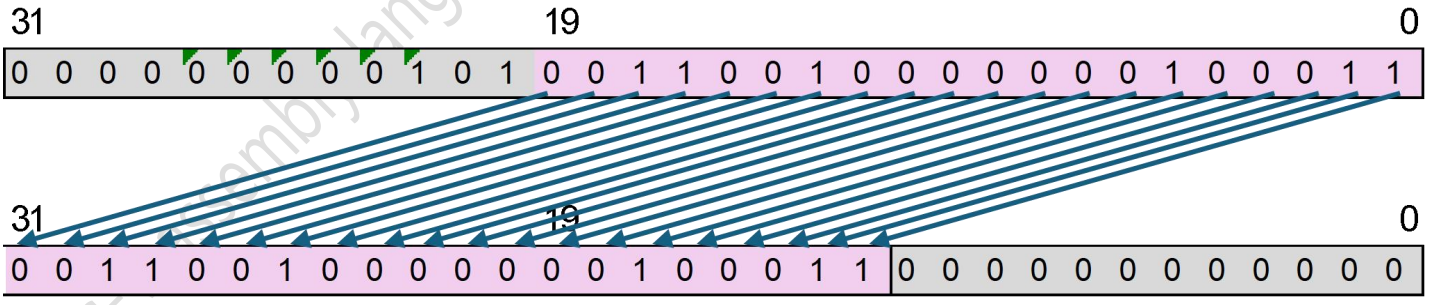
2.2.1.3.4 U-type instruction

The *U-Type* format is used by two instructions – `LUI` and `AUIPC`. There are 20 bits in the immediate field, which permits a larger range of immediate data. These 20 bits are shifted 12 places to the left and represent bits 31 through 12 of a destination register.

The lui instruction sets the lower 12 bits of the destination register to zeroes, as shown. An additional I-type instruction (12-bit immediate) is used to provide bits 11 through 0 of the destination to form a full 32-bit value.

AUIPC adds the 12-place, left-shifted immediate 20 bits to the program counter, placing the result into a destination register.

Figure 2- 2lui left shift of IMM bits into bits 31:12



2.2.1.3.4.1 AUIPC example

Assume that the program counter has the value 0x000100b0, the instruction `auipc t0,0x5a5a5` will add the immediate data 0x5a5a5000 to 0x000100b0, placing this 0x5a5b50b0 into5a5b register t0.

Table 2- 6 auipc example

Immediate value (left shifted by 12 places)	**5**	**a**	**5**	**a**	**5**	**0**	**0**	**0**
								+
Current program counter	0	0	0	1	0	0	b	0
Register t0	5	a	5	b	5	0	b	0

2.2.1.3.4.2 LUI example

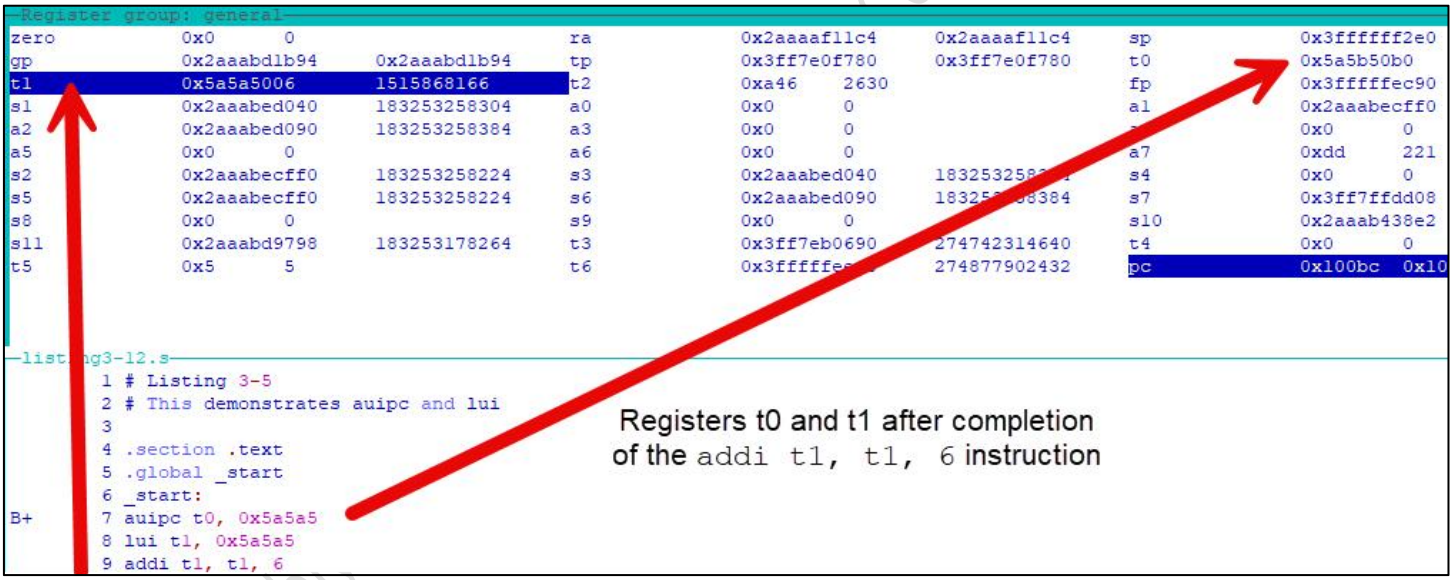
The instruction `lui t1,0x5a5a5` will add the immediate data 0x5a5a5000 to register t1.

Table 2- 7 LUI example

Immediate value (left shifted by 12 places)	**5**	**a**	**5**	**a**	**5**	**0**	**0**	**0**
Register t1	5	a	5	b	5	0	0	0

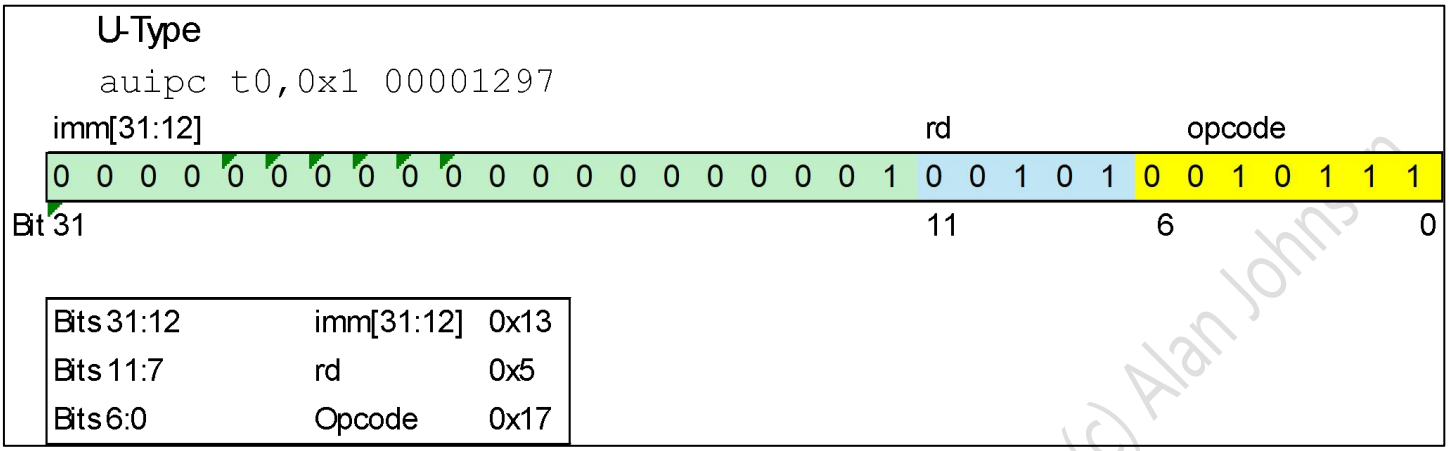
The trace in Figure 2- 3 shows the contents of the registers after program execution.

Figure 2- 3Tracing auipc and lui instructions



The format of AUIPC is shown in Table 2- 8 below.

Table 2- 8 Bit fields of the auipc U-Type instruction



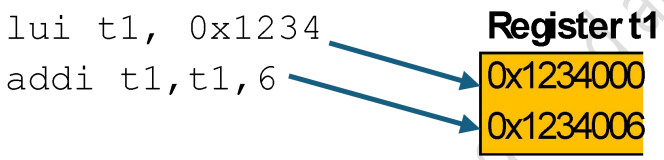
2.2.1.3.4.3 LUI with two instructions

The instructions to load a 32-bit value such as 0x1234006 into a register would look like:

```
lui t1, 0x1234      # Loads upper 20 bits
addi t1,t1, 6 # Loads lower 12 bits
```

The first instruction shifts the 20-bit value over 12 places and places zeros in the lower 12 bits. The second instruction adds the 12-bit value 0x006 to the current contents of t1 (0x1234000 + 006 = 0x1234006) and places the result in t1 as shown in Figure 2- 4.

Figure 2- 4Using LUI and ADDI to generate a 32-bit immediate value.



There is, however, an easier method by using the pseudo-instruction Load Immediate `LI`. Instead of using the two instructions shown above, the code using `LI` looks like:

```
li t1, 0x1234006.
```

This pseudo instruction could be translated into:

```
lui t1, 01234
addw t1, t1, 6
```

Pseudo-instructions are automatically translated by the assembler to one or more real machine instructions.

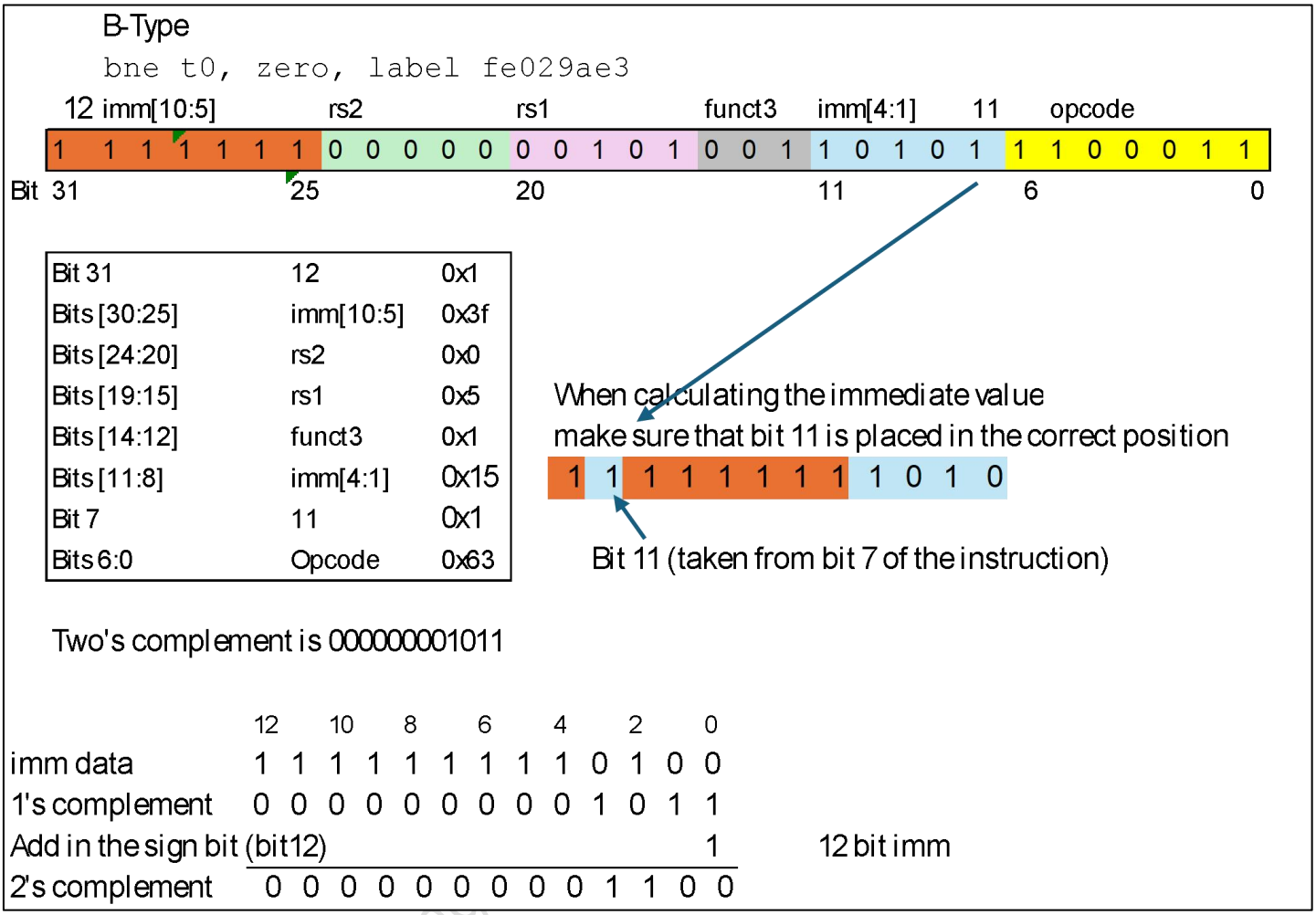
2.2.1.3.5 B-Type Instruction

The *B-Type* instruction is used with *conditional branches*. With this instruction, the immediate field has a range of 13-bits. This is achieved by setting the least significant bit to be zero and then substituting its bit position with bit 11 in the immediate field imm[4:1].

The location of the label `<putit>` is at address 0x100f8 and the branch instruction is located at address 0x10104.

```
10104:       fe029ae3                bne     t0,zero,100f8 <putit>
```

Table 2- 9Bit fields of the B-Type instruction



Since this is a backwards-pointing branch, the immediate field is a minus value; converting it using two's complement gives a value of 0xc or 12 places back.

2.2.1.3.6 J-Ttpe Instruction

Unconditional branches use the Jump and Link instruction (`JAL`). This is a *J-type* instruction. It is similar to the B-type instruction, with the immediate bits in disjoint fields to allow for more efficient decoding. In this example a `JAL` instruction is encountered at program counter address 0x100c0 and the instruction points to a label located at 0x100d4. The number of bytes to jump is encoded in the immediate bits as shown in Table 2- 10. The machine code produced is 0x014000ef. The opcode for the instruction is 0x6f and the destination register is x1 which is the return address register (`ra`).

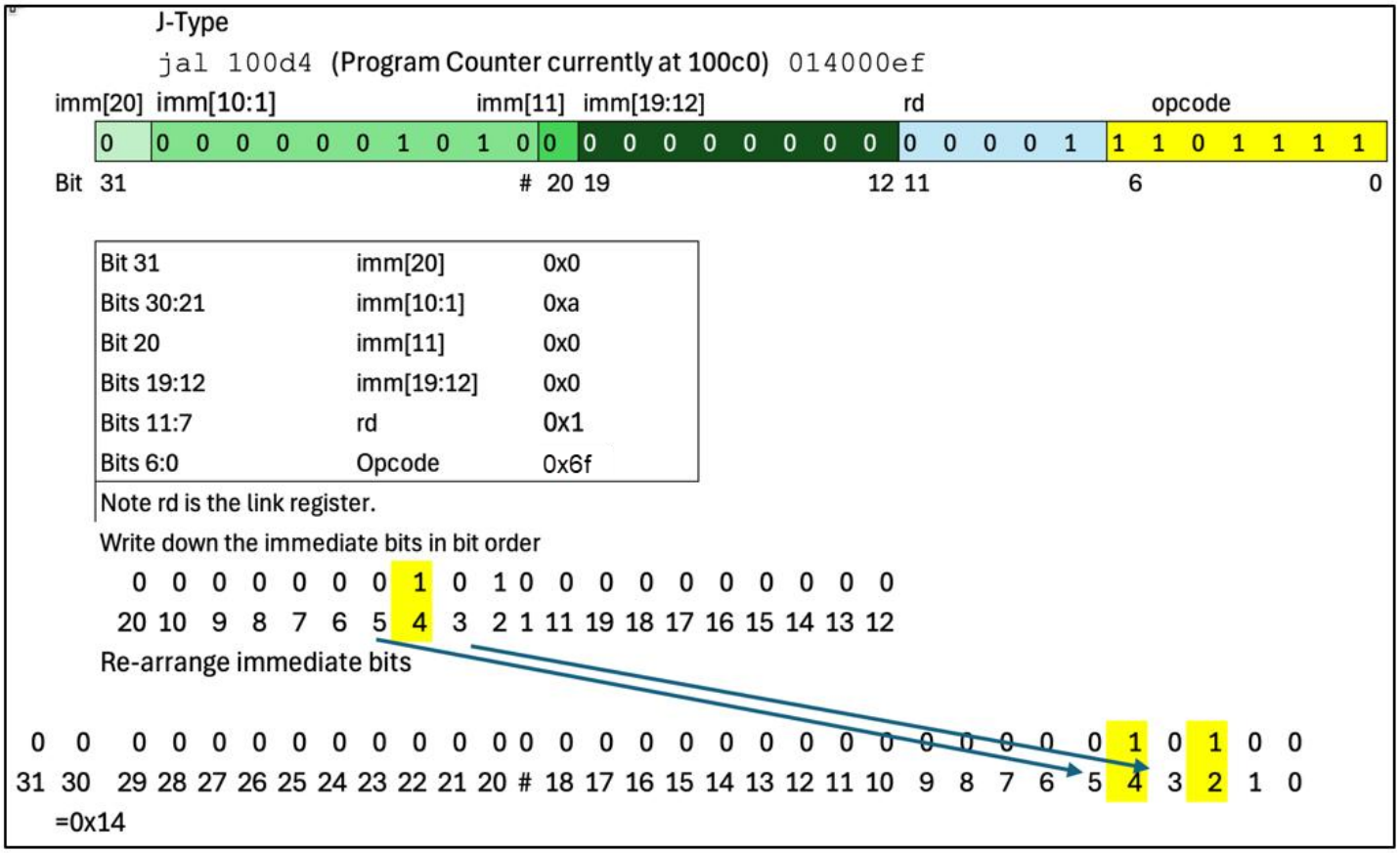
There is an aliased instruction `J` which uses the zero register instead of the ra register as shown below:

The aliased instruction `j 100b8 <quit>`

is encoded as:

```
jal     zero,100b8 <quit>
```

Table 2- 10 Bit fields of the J-Type instruction



More information regarding branch and jump instructions is covered in a dedicated chapter of the book.

2.2.2 Additional fields: funct3 and funct7

The opcode can be the same for different *related* instructions. The opcode is 7-bits wide, occupying bit positions 6:0. To take a specific example, the opcode 0110011 (0x36) refers to R-Type integer arithmetic and logical instructions. Each of these instructions is differentiated by the funct3 and funct7 fields. The numerical suffix refers to the number of bits used – funct3 is a three-bit field and funct7 is a seven-bit field. This is really a ten-bit field broken up into two subfields. It is non-contiguous to ensure consistency across the different types of instructions (aiding decoders) and to avoid waste by reusing the bits for different purposes, if a funct field is not required for that particular instruction type. The type defines which funct field(s) are in use. The funct3 field will always occupy bits 14:12 on any instruction that uses the field, and funct7 (used on the R-Type instruction) will occupy bit positions 31:25. The U and J-type instructions do not use the funct fields and will use these bits to specify a greater range of immediate bits. Table 2- 11 shows which of the funct fields are used with each instruction type.

Table 2- 11Funct field usage with instruction types.

Instruction Type	Funct3 field	Funct7 field
R-Type	Yes	Yes

I-Type	Yes	No
S-Type	Yes	No
B-Type	Yes	No
U-Type	No	No
J-Type	No	No

For R-Type instructions, the funct fields define the operation type. Some of these definitions are listed in Table 2- 12 below.

Table 2- 12Funct fields used for R-Type Integer instructions

Instruction	Opcode	Funct7	Funct3
ADD	0110011	0000000	000
SUB	0110011	0**1**00000	000
SRA	0110011	0**1**00000	101
SLL	0110011	0000000	001
SRL	0110011	0000000	101
AND	0110011	0000000	111
OR	0110011	0000000	110
XOR	0110011	0000000	100

Note that ADD and SUB have the same funct3 value; they are differentiated by bit 5 of funct7 and, similarly, SRL and SRA instructions.

2.3 Coding Tools

Chapter One gave a brief introduction to the assembly process. The tool that will be used for assembly is the GNU assembler (GAS). This utility is also used when compiling higher-level languages to provide intermediate code during the compilation process. It is part of the open-source GNU Binutils[21] collection. The binary tools include (amongst others) –

Table 2- 13GNU Tools associated with assembling and linking

Tool Name	Function
as	Assembler
ld	Linker

[21] *Use sudo apt install -y binutils - See https://www.gnu.org/software/binutils/ for more detail*

objdump	Disassembles and dumps object file information
make	Utility for assembling and linking multiple files, ignoring files that are up to date.

The candidate platforms suggested in this chapter include the tools listed in Table 2- 13. The GNU tools are applicable to a wide range of architectures, including Intel® and Arm®. The listings in this book have all been tested with the GNU assembler[22]. The GNU *toolchain*[23] also includes other programming tools, such as GNU Autotools and Bison for *parsing*.

The next section illustrates the use of all the tools listed in Table 2- 13

2.3.1 Editing files

The first stage in the assembly process is to edit the source files/ The assembly code is plaintext, so basic text editors such as *vi* or *nano* should be used. By convention, the file suffix is ".s", so a command to write a source program could be a command such as `vi testprogram.s`, which would edit an existing file or create a new one if it did not already exist. An example of a small assembly program is shown below:

Listing 2-1 Basic assembly code

```
.section .text
.global _start
_start:
addi t1, zero,6   # mov 6 into t1
addi t2, zero, 11 # mov 11 into t2
add t3, t1, t2      # add t2 and t1 result goes to t3
addi a7, x0, 93     # Call
ecall
```

The first line of code defines a label (`_start`) that marks the entry point of the program. The entry `.global`[24] is a *directive* to the assembler defining an action. Directives are not part of the actual machine code that will be produced, but will help the assembly process by providing instructions on how to control flow, define symbols, and reserve space, as well as other tasks. They also aid the coder in that they can define strings of text without having to refer to tables of *ASCII* codes. The code after the `_start` label is the actual code that will be assembled into RISC-V machine code.

The program moves two numbers, 6 and 11 into two registers (t1 and t2) and then adds them together, placing the result of the addition in register t3. The next two lines of code use operating system calls (*syscalls*) to gracefully exit the program.

2.3.1.1 System calls

System Calls (Syscalls) are service requests sent to the Operating Systems' kernel to perform a privileged task. These tasks include interacting with the hardware, file operations, memory management, and networking. When a syscall is

[22] *The GNU assembler is recommended for all the listings here.*

[23] *A toolchain is a collection of programming tools.*

[24] *Some listings may use ".globl" . Both forms are acceptable to the GNU assembler*

invoked, the system switches to a *privileged* mode, which executes tasks in a coordinated, standard manner. Syscalls are different across architectures and operating systems. With RISC-V systems running under Linux, the first step is to place the syscall code in register a7 and then use the `ECALL` instruction to request the function. The last two instructions of the code show how to use the *exit* system call.

2.3.1.1.1 Bare metal programming

Bare metal programming is a term used when the code interacts directly with the machine itself, it does not have the benefit of the Operating System support and so syscalls are unavailable. Bare metal programming is commonly used in *embedded* systems.

2.3.1.2 Sections

Assembly language source files are typically divided into *sections*. The sections used with RISC-V assembly files include the following.

Table 2- 14 Assembly language sections

Section Name	Purpose
.text	Contains the source-level instructions.
.data	Allocation of initialized variables.
.rodata	Holds constants or text strings that are read only.
.bss	Allocates uninitialized data buffers.

The following code shows the use of sections and how they are interacted with by registers.

Listing 2-2 Interacting with assembly sections.

```
# testprogram
.section .text
# The .text section contains the assembly language source instructions, omitting
# the prefix .section also works for .text, .data and .bss sections.
.global _start
_start:
/* This program illustrates the use of sections in RISC-V assembly.
It also shows how to interact with memory via load and store instructions.
Note this text is encapsulated using a multi-line comment.
The other comments using the # character are single-line comments */
.option norelax
# Interacting with .data section
# Good practice to use align
Object Dump.align 4
```

```
    lw a0, oneword      # Loads the content of the address at oneword into a0
.align 4
    lh a1, onehalf  # Loads the content of the address at onehalf into a1
.align 4
    lb a2, onebyte  # Loads the content of the address at onebyte into a2

# Interacting with .bss section
    la a3, buffer1
    sw a0, 0(a3)
    addi a7, x0, 93
    ecall

# Interacting with .rodata section
    lb a4, min
    lb a5, max

.data           # This section initializes variables.
oneword:        .word 2
onehalf:        .half 0xaa55
onebyte:        .byte 0x44

.section .rodata        # This section can hold constants and text strings
min:    .byte 32
max:    .byte 100

.bss            # This section allocates memory space for storage
buffer1: .space 100
```

This listing introduces align directives that are used to ensure that the bytes are kept on byte boundaries. Misalignment can hinder performance or cause illegal traps in some circumstances.

2.3.1.3 Introduced directives

The `.align` → alignment is specified as a power of two, and the `.balign` → alignment is specified in bytes.

Discussion of the directive .option norelax is deferred until section 3.6

The *data* section defines three variables, each occupying a different byte count. *BSS* allocated region(s) of memory. The rodata section holds constants that are not changed.[25] See Table 2- 14 for a summary.

2.3.2 Looking internally

Sections can be shown using the `objdump` command.

```
objdump -h listing2-2

listing2-2:     file format elf64-littleriscv

Sections:

Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000050  00000000000100f0  00000000000100f0  000000f0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000002  0000000000010140  0000000000010140  00000140  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00000007  0000000000011142  0000000000011142  00000142  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00000067  0000000000011149  0000000000011149  00000149  2**0
                  ALLOC
  4 .riscv.attributes 00000037  0000000000000000  0000000000000000  00000149  2**0
                  CONTENTS, READONLY
  5 .debug_aranges 00000030  0000000000000000  0000000000000000  00000180  2**4
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_info   0000002e  0000000000000000  0000000000000000  000001b0  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_abbrev 00000014  0000000000000000  0000000000000000  000001de  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line   0000006e  0000000000000000  0000000000000000  000001f2  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_str    0000003a  0000000000000000  0000000000000000  00000260  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

2.3.2.1.1 Virtual Memory Address and Load Memory Address

In the output of the objdump command given above, there are field headings *VMA* and *LMA*. These are the virtual and load memory addresses. The virtual memory address is the section address at runtime, and the load memory address is the location where the section is loaded.

[25] *Note that the constants could have been defined in the .data section with no adverse affect*

These locations are usually the same but can differ if ROM memory (LMA) needs to be re-located to writable RAM memory (VMA).

More information regarding objdump is listed in section 2.3.7.

2.3.3 Comments

Comments are ignored by the assembler but are important for maintaining code clarity. There are multi-line comments beginning with `/*` and ending with `*/` and single-line comments using the `#` character.

2.3.4 Assembling

Note high-level languages have an additional stage between editing and assembling – this is the *compilation* stage which will generate assembly code from a high-level language source code.

Later in[26] the book, the mixing of hybrid high-level languages and assembly code will be covered, but until then, only pure assembly language programming will be discussed.

Once the file has been edited, it can be assembled. The assembler will check for syntax[27] errors and if successful it will generate an object file. This is the main task of the assembler – *to generate machine code for the underlying processor architecture*. It is also responsible for translating RISC-V pseudo instructions into real machine code instructions. The object file uses the suffix ".o". The **G**NU assembler may be referred to as GAS!

The command to assemble a program is shown below:

```
as -o testprogram.o testprogram.s
```

Note the order of the files:the object file name is given first, followed by the source name.

When initially developing programs, it is normal to include extra information to assist with the debugging process. Once the code is ready for final release, this extra information is removed. The command to include debugging information is:

```
as -g -o testprogram.o testprogram.s
```

Including the debugging data increases the size of the code. The assembler ignores the comments, which are only used for human clarification purposes and have no meaning to the processor. Once the code has been translated into machine code, it is not yet in an executable state. Along with the actual machine code, a number of *symbol* references may be defined. These may be references to symbols defined in other object files that the current source program has no access to.

2.3.5 Linker

The linker's role is to produce code that can be executed by the system; most large programs are not standalone but instead consist of a number of smaller programs or library files. The linker "joins" these programs together and

[26] *See Page 184.*

[27] *Note logic/flow error checking is largely the coder's responsibility.*

generates the final executable. In addition, the linker has the responsibility of resolving the symbol references. It will also perform optimization.

Other considerations include integration within the file system. Linux programs use the *Executable and Linkable format* (ELF). This format is portable, supporting a wide range of platforms. ELF files consist of headers and sections to aid in mapping the program into memory. The `readelf` utility analyzes the ELF format. The ELF header can be shown with the command `readelf -h listing2-2` as shown below:

```
$ readelf -h listing2-2
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x100f0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          1640 (bytes into file)
  Flags:                             0x4, double-float ABI
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         14
  Section header string table index: 13
```

The program will commence execution at memory address 0x100f0

To produce an executable from the `listing2-2.o` file, use the command –

```
ld -o listing2-2 listing2-2.o
```

2.3.5.1 Linker Scripts

Linker scripts are used to describe memory allocation maps and are more commonly customized in embedded systems. They are text files.

The command `ld -verbose` lists the contents of the default linker script on Debian-based Linux. For our purposes, this is for informational only and will not be discussed further.

```
GNU ld (GNU Binutils for Debian) 2.40
Supported emulations:
elf64lriscv
elf64lriscv_lp64f
elf64lriscv_lp64
elf32lriscv
elf32lriscv_ilp32f
elf32lriscv_ilp32
elf64briscv
elf64briscv_lp64f
elf64briscv_lp64
elf32briscv
elf32briscv_ilp32f
elf32briscv_ilp32
using internal linker script:
==================================================
/* Script for -z combreloc */
/* Copyright (C) 2014-2023 Free Software Foundation, Inc.
Copying and distribution of this script, with or without modification,
. . .
/* DWARF 3.  */
.debug_pubtypes 0 : { *(.debug_pubtypes) }
.debug_ranges   0 : { *(.debug_ranges) }
/* DWARF 5.  */
.debug_addr     0 : { *(.debug_addr) }
.debug_line_str 0 : { *(.debug_line_str) }
.debug_loclists 0 : { *(.debug_loclists) }
.debug_macro    0 : { *(.debug_macro) }
.debug_names    0 : { *(.debug_names) }
.debug_rnglists 0 : { *(.debug_rnglists) }
.debug_str_offsets 0 : { *(.debug_str_offsets) }
```

```
.debug_sup      0 : { *(.debug_sup) }

.gnu.attributes 0 : { KEEP (*(.gnu.attributes)) }

/DISCARD/ : { *(.note.GNU-stack) *(.gnu_debuglink) *(.gnu.lto_*) }
```

On the Debian system used here `(Linux starfive 6.6.20-starfive #41SF SMP Fri Sep 20 17:48:26 CST 2024 riscv64 GNU/Linux`) the linker scripts are located at `/lib/riscv64-linux-gnu/ldscripts/ -`

```
lf32briscv_ilp32f.x                             elf32briscv_ilp32.xdc                elf32briscv.xe
elf32lriscv_ilp32f.xsceelf32lriscv_ilp32.xw       elf64briscv_lp64f.xce       elf64briscv_lp64.xdw
elf64briscv.xs          elf64lriscv_lp64f.xswe  elf64lriscv.xbn

elf32briscv_ilp32f.xbn   elf32briscv_ilp32.xdce  elf32briscv.xn

. . .
```

2.3.6 GDB – The GNU Debugger

GDB is used to view program flow. The code can be run “one step (instruction) at a time” as a teaching tool to promote understanding of program execution. It does this by displaying register and memory contents, along with the line of source code being executed. The tool is invaluable for coders looking to track down more elusive issues such as unexpected results. By single-stepping through the code, the exact location where the error occurs can be readily identified. As mentioned earlier, the assembler command `as` uses the `-g` switch to generate debugging information. The debugger can be launched by the `gdb` command.

GDB can be installed on Debian systems with the command `sudo apt install -y gdb`

```
sudo apt install -y gdb

Reading package lists... Done

Building dependency tree... Done

Reading state information... Done

The following additional packages will be installed:

  libbabeltrace1  libboost-regex1.74.0  libc6-dbg  libdebuginfod-common  libdebuginfod1  libsource-
highlight-common libsource-highlight4v5

Suggested packages:

  gdb-doc gdbserver

The following NEW packages will be installed:

  gdb libbabeltrace1 libboost-regex1.74.0 libc6-dbg libdebuginfod-common libdebuginfod1 libsource-
highlight-common libsource-highlight4v5

0 upgraded, 8 newly installed, 0 to remove and 54 not upgraded.

Need to get 11.3 MB of archives.

After this operation, 24.9 MB of additional disk space will be used.

. . .
```

To illustrate GDB in action, issue the command

```
gdb testprogram
```

```
$ gdb testprogram
GNU GDB (Debian 13.2-1) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
. . .
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from testprogram...
(gdb) list
.section .text
.global _start
start:
addi t1, zero,6   # mov 6 into t1
addi t2, zero, 11 # mov 11 into t2
add t3, t1, t2      # add t2 and t1 result goes to t3
addi a7, x0, 93
ecall
(gdb) b 1
Breakpoint 1 at 0x100b0: file testprogram.s, line 6.
(gdb) run
Starting program: /home/alan/asm/misc/testprogram
Breakpoint 1, _start () at testprogram.s:6
6             addi t1, zero,6   # mov 6 into t1
(gdb) n
7             addi t2, zero, 11 # mov 11 into t2
(gdb) n
8             add t3, t1, t2      # add t2 and t1 result goes to t3
(gdb) n
9             addi a7, x0, 93
(gdb) i r t1
t1             0x6  6
(gdb) i r t2
t2             0xb  11
```

```
(gdb) i r t3
t3             0x11 17
(gdb) q
A debugging session is active.
Inferior 1 [process 8652] will be killed.
Quit anyway? (y or n) y
```

Table 2- 15 lists some of the more commonly used GDB[28] commands

Table 2- 15 Commonly used GDB commands

Command	Meaning
List	List the source assembly file
B 1	Sets a stopping point known as a breakpoint once the program runs, a number can be used or a label such as b _start
Run	Starts the program and halts at the first breakpoint (if any has been set).
N(ext)	Advances to the next (n)line(s)[29] of code, by-passing sub-routines.
S(tep)	Steps to the next (n)line(s) of code, entering subroutines.
I(nfo) t1	Shows the contents of register t1
I(nfo) t2	Shows the contents of register t2
I(nfo) t3	Shows the contents of register t3
Q(uit)	Exits the program

GDB will be covered in more detail as the document progresses; in addition, it will function as the primary learning tool to illustrate program flow and how each of the instructions work[30].

2.3.7 Objdump

The objdump utility is helpful with reverse engineering and understanding object code. The code can be disassembled to show the original source instructions using the -d switch, for example:

```
$ objdump -d listing2-2
listing2-2:     file format elf64-littleriscv
```

[28] *A more comprehensive list is found in the appendix.*

[29] *Default is one line*

[30] *The reader is encouraged to use GDB to step through the program listings.*

```
Disassembly of section .text:


00000000000100f0 <_start>:
   100f0:       00001517                auipc   a0,0x1
   100f4:       05252503                lw      a0,82(a0) # 11142 <__DATA_BEGIN__>
   100f8:       00000013                nop
   100fc:       00000013                nop
   10100:       00001597                auipc   a1,0x1
   10104:       04659583                lh      a1,70(a1) # 11146 <onehalf>
   10108:       00000013                nop
   1010c:       00000013                nop
   10110:       00001617                auipc   a2,0x1
   10114:       03860603                lb      a2,56(a2) # 11148 <onebyte>
   10118:       00001697                auipc   a3,0x1
   1011c:       03168693                addi    a3,a3,49 # 11149 <__SDATA_BEGIN__>
   10120:       00a6a023                sw      a0,0(a3)
   10124:       05d00893                li      a7,93
   10128:       00000073                ecall
   1012c:       00000717                auipc   a4,0x0
   10130:       01470703                lb      a4,20(a4) # 10140 <min>
   10134:       00000797                auipc   a5,0x0
   10138:       00d78783                lb      a5,13(a5) # 10141 <max>
   1013c:       00000013                nop
```

The option `-M no-aliases` allows us to see how the assembler *translated* the pseudo instruction `li` –

```
$ objdump -d -M no-aliases listing2-2


listing2-2:     file format elf64-littleriscv
Disassembly of section .text:
00000000000100f0 <_start>:
   100f0:       00001517                auipc   a0,0x1
   100f4:       05252503                lw      a0,82(a0) # 11142 <__DATA_BEGIN__>
   100f8:       00000013                addi    zero,zero,0
   100fc:       00000013                addi    zero,zero,0
```

```
   10100:       00001597                auipc   a1,0x1
   10104:       04659583                lh      a1,70(a1) # 11146 <onehalf>
   10108:       00000013                addi    zero,zero,0
   1010c:       00000013                addi    zero,zero,0
   10110:       00001617                auipc   a2,0x1
   10114:       03860603                lb      a2,56(a2) # 11148 <onebyte>
   10118:       00001697                auipc   a3,0x1
   1011c:       03168693                addi    a3,a3,49 # 11149 <__SDATA_BEGIN__>
   10120:       00a6a023                sw      a0,0(a3)
   10124:       05d00893                addi    a7,zero,93
   10128:       00000073                ecall
   1012c:       00000717                auipc   a4,0x0
   10130:       01470703                lb      a4,20(a4) # 10140 <min>
   10134:       00000797                auipc   a5,0x0
   10138:       00d78783                lb      a5,13(a5) # 10141 <max>
   1013c:       00000013                addi    zero,zero,0
```

Since the immediate data was small (less than one byte) `li` was achieved using the single instruction `addi`.

2.3.8 Make

The commands that have been used so far for assembling and linking (`as,ld`) have worked well enough for our situation; however, when multiple files are involved, it is normal to use a build tool to accomplish this. The *make* utility keeps track of what has been done and will only apply actions to the changed portions. The instructions are conveyed to the utility using a *makefile*. The `makefile` below can be used to assemble and link the program `testprogram.s`

```
Simple makefile

testprogram: testprogram.o
	ld -o testprogram testprogram.o


testprogram.o: testprogram.s
	as -o testprogram.o testprogram.s
```

The line at the top denotes the *target* file, which *depends* on the *object* file, which in turn depends on the *source* file. The rules on how to create the target file are shown above, so the flow is →

- Create the target file (`testprogram`) from the object file (`testprogram.o`) which is created from the source file (`testprogram.s`). The first target (here `testprogram`) is termed the *default goal*.

The makefile is invoked by the command:

```
make testprogram

as -o testprogram.o testprogram.s

ld -o testprogram testprogram.o
```

or in this case, simply enter: -

```
make

make: 'testprogram' is up to date.
```

Note use of tab characters for indentation in the `makefile`.

The next example assembles and links two programs into a single executable file.

```
OBJECTS = program1.o program2.o

all: myprogram

%.o: %.s
	as $< -g -o $@

myprogram: $(OBJECTS)
	ld -o myprogram $(OBJECTS)
```

This example will allow the target to be passed to the `makefile`:-

```
TARGETFILE = $(targetfile)

print: $(TARGETFILE).o

	ld -o $(TARGETFILE) $(TARGETFILE).o

$(TARGETFILE).o: $(TARGETFILE).s

	as -g -o $(TARGETFILE).o $(TARGETFILE).s

$ make targetfile=print

make: 'print' is up to date.

$ ls

makefile  print  print.o  print.s
```

Try the next script -

```
TARGETFILE = $(targetfile)

$(TARGETFILE): $(TARGETFILE).o

	@echo "Now linking $(TARGETFILE).o to $(TARGETFILE)"

	ld -o $(TARGETFILE) $(TARGETFILE).o

$(TARGETFILE).o: $(TARGETFILE).s

	@echo "Now assembling $(TARGETFILE).s to $(TARGETFILE).o with debug option"

	as -g -o $(TARGETFILE).o $(TARGETFILE).s
```

Invoke with `make targetfile=<filename>.`

2.4 Choosing a candidate platform

2.4.1 Hardware Platforms

Low-cost RISC-V hardware is available today, some RV64 hardware platforms that seem to work well[31] are:

- VisionFive2 RISC-V Single Board Computer, StarFive JH7110 Processor with Integrated 3D GPU, 8GB Memory
  - `starfivetech.com/en`
- LicheePi 4A 64bit LPDDR4X 16GB RISC-V Single Board Computer.
  - `https://wiki.sipeed.com/hardware/en/lichee/th1520/lp4a.html`
- BananaPi BPI-F3
  - `https://docs.banana-pi.org/en/BPI-F3/BananaPi_BPI-F3`
- Milk-V Jupiter (Mini ITX form factor)
  - `Milk-V Jupiter | RISC-V PC for Everyone`

2.4.2 Emulation and Simulation

An alternative is to use RISC-V emulation courtesy of QEMU which is an open-source emulator and virtualizer. See https://www.qemu.org/docs/master/ for more information.

Installation is covered in the next section.

"""Cross compilation is another option, which is covered later.

2.4.2.1 Configuring a QEMU-Based Virtual Machine

Note that if using physical hardware, the following steps can be skipped (if desired).

Refer to the link –

Architectures/RISC-V/QEMU - Fedora Project Wiki (https://fedoraproject.org/wiki/Architectures/RISC-V/QEMU) for the most up to date installation details.

2.4.2.1.1 Install Qemu

This emulator is installed on an X86 platform running Fedora Linux. Use the following command to install and configure:

```
sudo dnf install \
  libvirt-daemon-driver-qemu \
  libvirt-daemon-driver-storage-core \
  libvirt-daemon-driver-network \
```

[31] *Not all support Single Board Computers support vector instruction. The code in the vector chapter of this book was tested on the BananaPi BF3.*

```
    libvirt-daemon-config-network \
    libvirt-client \
    virt-install \
    qemu-system-riscv-core \
    edk2-riscv64
[sudo] password for fedorauser:
Updating and loading repositories:
Repositories loaded.
gpg: directory '/root/.gnupg' created
gpg: /root/.gnupg/trustdb.gpg: trustdb created
Package "libvirt-daemon-driver-qemu-11.0.0-2.fc42.x86_64" is already installed.
Package "libvirt-daemon-driver-storage-core-11.0.0-2.fc42.x86_64" is already installed.
Package "libvirt-daemon-driver-network-11.0.0-2.fc42.x86_64" is already installed.
Package "libvirt-daemon-config-network-11.0.0-2.fc42.x86_64" is already installed.
Package "libvirt-client-11.0.0-2.fc42.x86_64" is already installed.
Package     Arch        Version       Repository          Size
Installing:
 edk2-risc noarch 20250221-8.fc42 fedora  17.6 MiB
 qemu-system-riscv-core  x86_64 2:9.2.3-1.fc42
. . .
```

2.4.2.1.2 Set up access and default URI

```
$ sudo usermod -a -G libvirt $(whoami)
$ mkdir -p ~/.config/libvirt && \
echo 'uri_default="qemu:///system"' > ~/.config/libvirt/libvirt.conf
[fedorauser@fedora ~]$ sudo reboot
```

2.4.2.1.3 Get the image

```
$   wget   https://dl.fedoraproject.org/pub/alt/risc-v/release/42/Cloud/riscv64/images/Fedora-Cloud-Base-Generic-42.20250911-2251ba41cdd3.riscv64.qcow2
. . .
```

2.4.2.1.4 Re-locate and rename the image

```
$ sudo cp Fedora-Cloud-Base-Generic-42.20250911-2251ba41cdd3.riscv64.qcow2 /var/lib/libvirt/images/fedora-riscv.qcow2
```

Set up the environment and .yml file

```
$ mkdir ~/riscv
```

```
$ cd riscv
```

Edit and populate the file `user-data.yaml`

```
$ vi user-data.yaml
#cloud-config
password: linux
chpasswd:
  expire: false
runcmd:
  - touch /etc/cloud/cloud-init.disabled
```

2.4.2.1.5 Set up the VM

Configure parameters, editing as required, such as RAM and CPU parameters

```
$ virt-install \
    --import \
    --name fedora-riscv \
    --osinfo fedora-rawhide \
    --arch riscv64 \
    --cpu mode=maximum \
    --vcpus 4 \
    --ram 8192 \
    --boot uefi \
    --disk path=/var/lib/libvirt/images/fedora-riscv.qcow2 \
    --network default \
    --tpm none \
    --graphics none \
    --controller scsi,model=virtio-scsi \
    --cloud-init user-data=user-data.yaml
. . .
Fedora Linux 42 (Cloud Edition)
Kernel 6.13.0-0.rc4.36.0.riscv64.fc42.riscv64 on riscv64 (ttyS0)
enp1s0: 192.168.122.132 fe80::5054:ff:fe43:927a
localhost login: [ 108.371505] cloud-init[982]: Cloud-init v. 24.2 running 'modules:final' at Thu,
15 May 2025 16:23:09 +0000. Up 107.79 seconds.
ci-info: no authorized SSH keys fingerprints found for user fedora.
```

```
<14>May 15 16:23:11 cloud-init: ##########################

<14>May 15 16:23:11 cloud-init:--BEGIN SSH HOST KEY FINGERPRINTS-

<14>May    15    16:23:11    cloud-init:    256    SHA256:WerTtl94f5Use//HZikpuOTZyJzztfVWhGBhP0McjZU
root@localhost (ECDSA)

<14>May    15    16:23:11    cloud-init:    256    SHA256:FEOBA1tWS12IOewDzRywk0HOjkqZ2x66Rx++4LHkwO8
root@localhost (ED25519)

. . .

[ 109.684931] cloud-init[982]: Cloud-init v. 24.2 finished at Thu, 15 May 2025 16:23:11 +0000.
Datasource DataSourceNoCloud [seed=/dev/sr0][dsmode=net]. Up 109.56 seconds

Log on with username "fedora" and password "linux"

localhost login: fedora

Password:
```

2.4.2.1.6 Check the architecture -

```
[fedora@localhost ~]$ lscpu

Architecture: riscv64

Byte Order: Little Endian

CPU(s): 4

On-line CPU(s) list: 0-3

Vendor ID: 0x0

Model name: -

CPU family: 0x0

Model: 0x0

Thread(s) per core: 1

Core(s) per socket: 4

. . .

$ cat /proc/cpuinfo

processor       : 0

hart            : 0

isa                                                                                                 :
rv64imafdcvh_zicbom_zicbop_zicboz_ziccrse_zicntr_zicond_zicsr_zifencei_zihintntl_zihintpause_zihpm_
zimop_zaamo_zabha_zacas_zalrsc_zawrs_zfa_zfbfmin_zfh_zfhmin_zca_zcb_zcd_zcmop_zba_zbb_zbc_zbkb_zbkc
_zbkx_zbs_zknd_zkne_zknh_zkr_zkt_zksed_zksh_ztso_zvbb_zvbc_zve32f_zve32x_zve64d_zve64f_zve64x_zvfbf
min_zvfbfwma_zvfh_zvfhmin_zvkb_zvkg_zvkned_zvknha_zvknhb_zvksed_zvksh_zvkt_smaia_smmpm_smnpm_smstat
een_ssaia_sscofpmf_ssnpm_sstc_svadu_svinval_svnapot_svpbmt_svvptc

mmu             : sv57

mvendorid       : 0x0

marchid         : 0x0
```

```
mimpid          : 0x0

hart isa        :

. . .
```

Note the system has floating-point and vector support!

2.4.2.1.7 Check RAM size

```
$ free -m

total used free shared buff/cache available

Mem: 7911 348 7549 0 164 7562

Swap: 7910 0 7910
```

2.4.2.1.8 Check kernel release

```
[fedora@localhost ~]$ uname -a

Linux localhost 6.16.4-200.0.riscv64.fc42.riscv64 #1 SMP PREEMPT_DYNAMIC Fri Aug 29 08:42:48 EDT
2025 riscv64 GNU/Linux
```

2.4.2.1.9 Test the coding environment

2.4.2.1.9.1 Install tools

```
$ sudo dnf install binutils

. . .

Running transaction

[1/3]                                        Verify                                        package
files
                                  . . .100% |   2.4 MiB/s |  24.3 MiB |  00m10s

. . .

$ sudo dnf install -y gdb
```

Create a small assembly file

```
vi test.s

.global _start

_start:

la a1, hellorisc

addi a2, x0, 14

addi a7, x0, 64

ecall

addi a0, x0, 0

addi a7, x0, 93

ecall
```

```
.data

hellorisc:.ascii "Hello RISC-V!\n"
```

2.4.2.1.9.2 Assemble, link and execute

```
$ as -g -o test.o test.s

[fedora@localhost ~]$ ls

test.o test.s

[fedora@localhost ~]$ ld -o test test.o

[fedora@localhost ~]$ chmod 777 test

[fedora@localhost ~]$ ./test

Hello RISC-V!
```

2.4.2.1.10 Shutdown and restarting

Shutdown the machine with `$ sudo poweroff`

Restart with `$ virsh start fedora-riscv --console`

If `--console` is omitted then systems can be simply shutdown with the `virsh` command – `$ virsh shutdown <name>.` The active VMs can be shown with the `virsh list` command –

```
$ virsh list
 Id   Name           State
--------------------------
 1    fedora-riscv   running
$ virsh shutdown fedora-riscv
Domain 'fedora-riscv' is being shutdown
```

Shutdown all running systems using the script below -

```
for i in `virsh list | grep running | awk '{print $2}'`; do virsh shutdown $i; done
```

2.4.2.1.11 Remove a VM

```
$ virsh list --all

 Id   Name            State

--------------------------------

 -    fedora1-riscv   shut off


$ virsh dumpxml --domain fedora1-riscv | grep 'source file'

      <source file='/var/lib/libvirt/images/fedora-riscv.qcow2'/>

$ virsh undefine --nvram fedora1-riscv

Domain 'fedora1-riscv' has been undefined

$ virsh list --all

 Id   Name   State

--------------------
```

2.4.2.2 Grow the virtual disk capacity

On the host machine, - the virtual disk image can be expanded with

```
sudo qemu-img resize /var/lib/libvirt/images/fedora-riscv.qcow2 +20G


Image resized.
```

This adds 20G capacity to the existing image. The next task is to boot the image and grow a partition (here partition3 will be expanded) with the `fdisk` utility provided by the virtual machine

```
[fedora@localhost ~]$ sudo fdisk /dev/vda
Welcome to fdisk (util-linux 2.40.4).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.
. . .
Command (m for help): p
Disk /dev/vda: 25 GiB, 26843545600 bytes, 52428800 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: D8EE907C-0F17-41BA-BBEC-8A0DA4FB0950
Device Start End Sectors Size Type
/dev/vda1 2048 206847 204800 100M EFI System
/dev/vda2 206848 2254847 2048000 1000M Linux extended boot
/dev/vda3 2254848 10485726 8230879 3.9G Linux root (RISC-V-64)
Command (m for help): e
Partition number (1-3, default 3):
New <size>{K,M,G,T,P} in bytes or <size>S in sectors (default 23.9G):
Partition 3 has been resized.
Command (m for help): p
Disk /dev/vda: 25 GiB, 26843545600 bytes, 52428800 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
```

```
Disk identifier: D8EE907C-0F17-41BA-BBEC-8A0DA4FB0950
Device Start End Sectors Size Type
/dev/vda1 2048 206847 204800 100M EFI System
/dev/vda2 206848 2254847 2048000 1000M Linux extended boot
/dev/vda3 2254848 52428766 50173919 23.9G Linux root (RISC-V-64)
Command (m for help): w
The partition table has been altered.
Syncing disks.
```

The `fdisk` utility has expanded the partition to 24G compared to the previous capacity of 4GB.

Verify from the Operating System -

```
[fedora@localhost ~]$ lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINTS
sr0 11:0 1 1024M 0 rom
zram0 251:0 0 7.7G 0 disk [SWAP]
vda 252:0 0 25G 0 disk
├─vda1 252:1 0 100M 0 part /boot/efi
├─vda2 252:2 0 1000M 0 part /boot
└─vda3 252:3 0 23.9G 0 part /var
/home
/
```

2.4.2.2.1 Optional activities

2.4.2.2.1.1 Add network tools and gdb

```
$ sudo dnf install -y net-tools
$ sudo dnf install -y gdb
```

2.4.2.3 Updating Fedora

```
$ sudo dnf upgrade --best
Fedora RISC-V                          4.1 kB/s | 3.8 kB     00:00
Dependencies resolved.
================================================================================
Package                Arch    Version                   Repository    Size
================================================================================
Installing:
iwlegacy-firmware    noarch  20230804-153.fc38       fedora-riscv 140
```

```
. . .
```

2.4.2.4 Removing QEMU images

```
$ virsh list --all
 Id   Name    State
-----------------------
 -    riscv   shut off


$ virsh undefine riscv
error: Failed to undefine domain 'riscv'
error: Requested operation is not valid: cannot undefine domain with nvram
```

If the above error is encountered use the command below:

```
$ virsh undefine riscv --nvram
Domain 'riscv' has been undefined


$ virsh list --all
 Id   Name   State
--------------------
```

2.4.2.5 Simulators

Simulators are as the name suggests, programs that run on the native system, providing the functionality of a target system. They can normally be run online or perhaps under the control of an environment such as Java. Two such programs are:

- CPUlator
- RARS

2.4.2.5.1 CPUlator

CPUlator[32] is an online simulator that supports RV32. The simulator is an excellent tool for debugging as it provides *single stepping* through the code, as well as showing memory, registers and code disassembly.

To get started, select the system to be emulated (here RISC-V RV32) as shown, and enter the code. The code can be made executable in the simulator by selecting <Compile and Load> as shown in Figure 2-6. The code can be executed one line at a time by selecting <Step Into>. Each step will show changes to the register and memory contents.

[32] *The URL for CPUlator is https://cpulator.01xz.net/*

Figure 2- 5 CPUlator home page

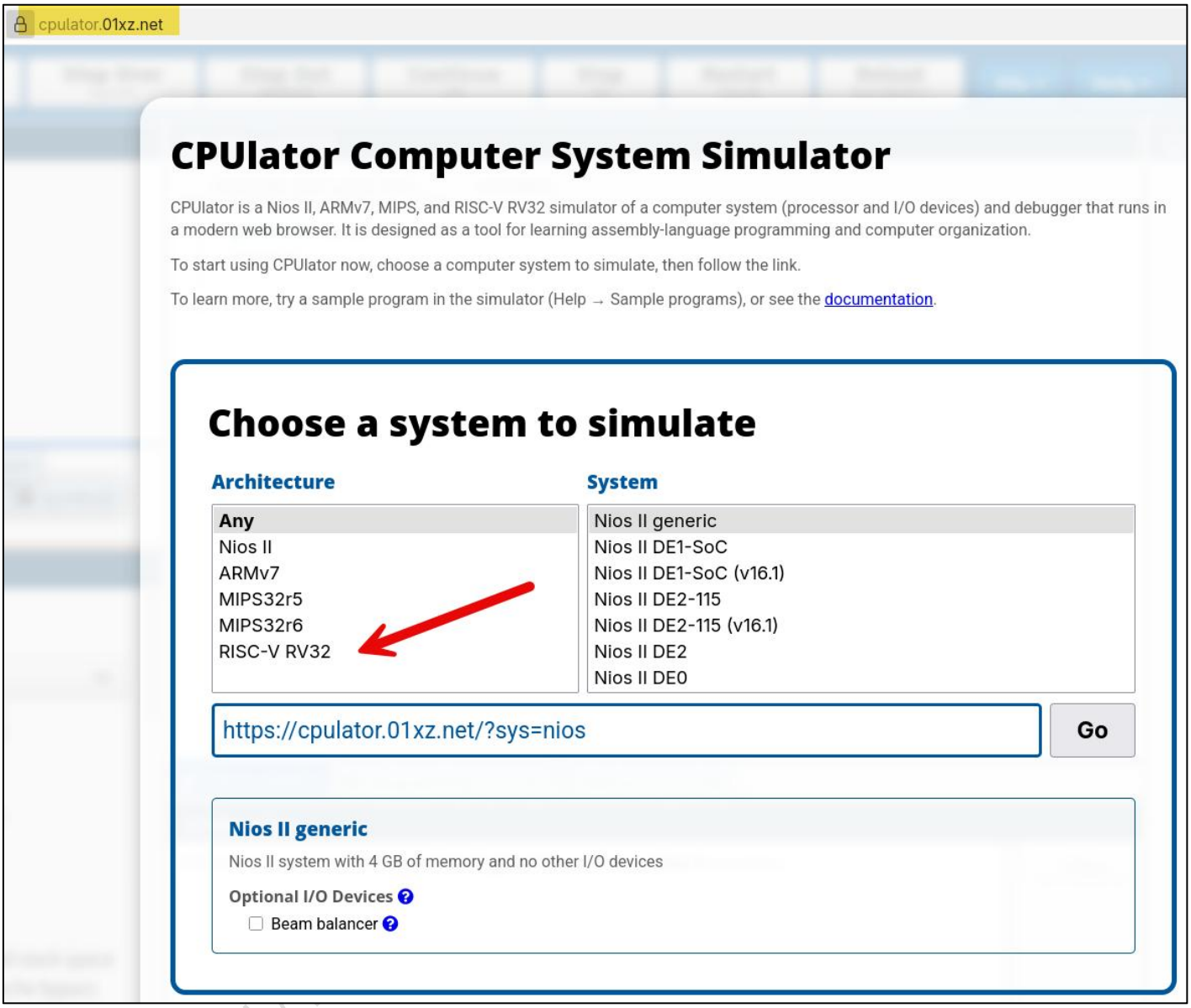
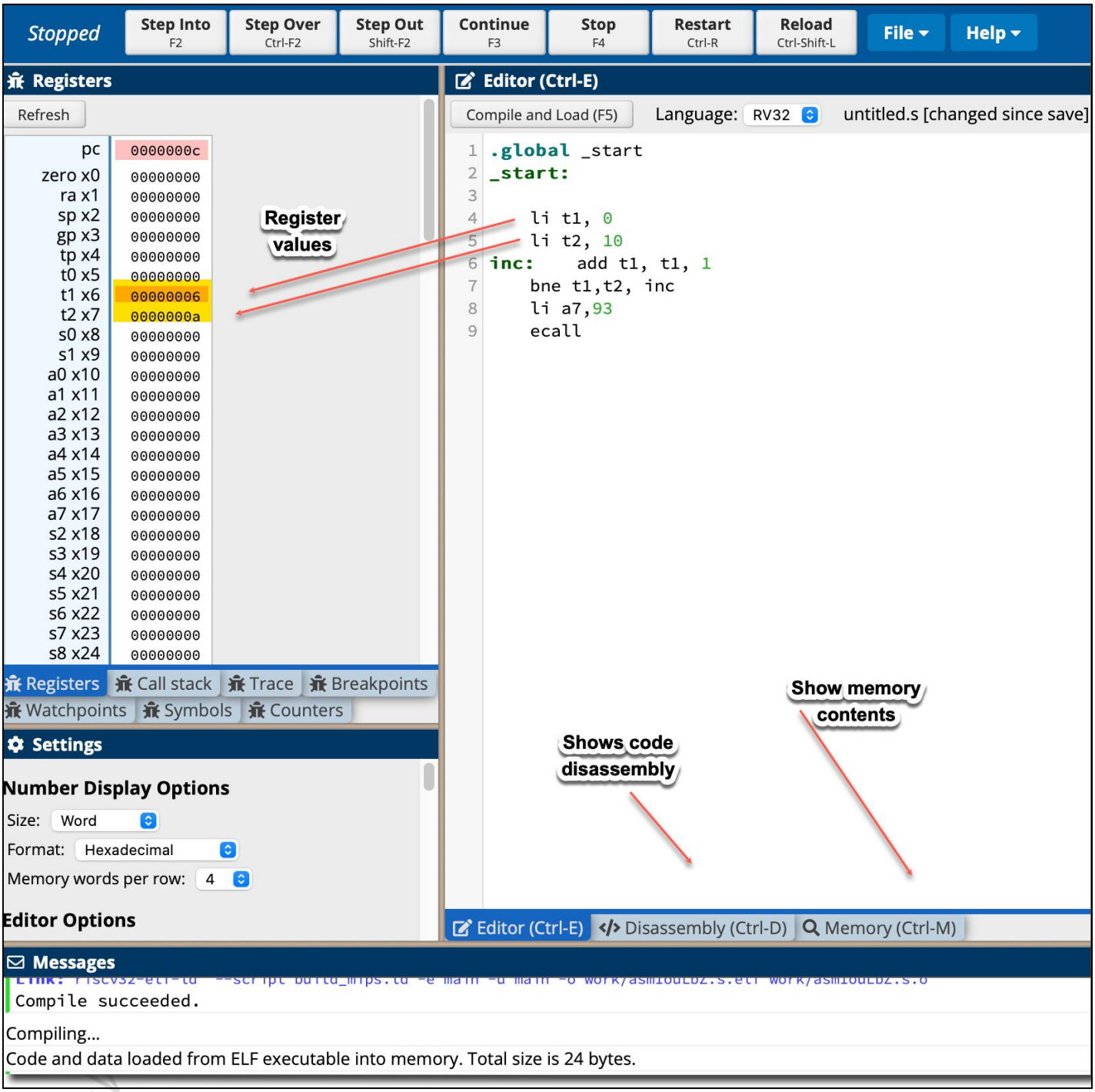


Figure 2- 6 Compiling and executing code with CPUlator

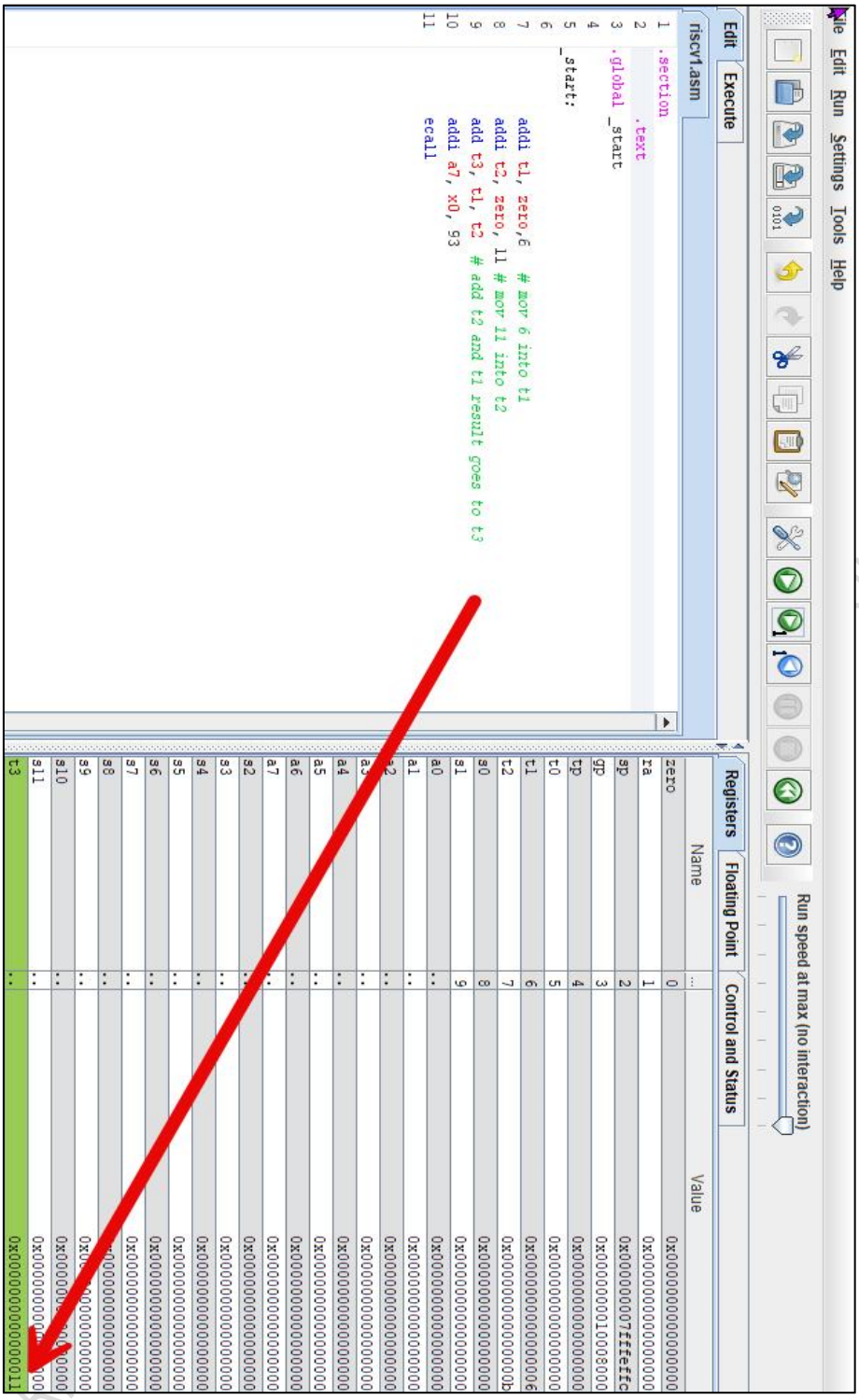


2.4.2.5.2 RARS

RARS[33] stands for RISC-V Assembler and Runtime Simulator. It is also an excellent tool for learning RISC-V assembly language.

[33] *The URL for RARS is https://github.com/TheThirdOne/rars/releases/tag/continuous*

Figure 2- 7RARS Execution screen



Refer to the URL in the footnote for more information on CPUlator and RARS.

In this example the downloaded RARS version was `rars_3897cfa.jar` as shown in Figure 2- 8. To run, execute `java -jar rars_3897cfa.jar`.

Figure 2- 8 Downloading RARS

The following link lists more simulators – https://www.riscvschool.com/risc-v-simulators/

2.4.3 Using strace

The *strace* utility can be used to monitor which syscalls have been invoked by a particular program or process: -

```
$ strace -c ./print
Hello again!
% time     seconds  usecs/call     calls    errors syscall
------------------------------------------------------------
0.00    0.000000           0         1           write
0.00    0.000000           0         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000           0         2           total
```

`Strace`, here shows that the syscalls `write` and `exit` were invoked once.

2.5 RISC-V Instructions Covered in Chapter 2

1. **ADDI** Add Immediate Example: addi t2, zero, 11
2. **ADD** Add (register-to-register) Example: add t3, t1, t2
3. **ECALL** – Environment call (used for syscalls)
4. **LUI** – Load Upper Immediate
5. **AUIPC** – Add Upper Immediate to PC
6. **sw** – Store Word
7. **JAL / JALR** – Jump and Link / Jump and Link Register (implicitly discussed in context of control transfer)

2.6 Exercises for chapter 2

1. What qualifier would you add to the `as` command to embed debug information?
2. What is the purpose of a linker?
3. How many registers are available for general-purpose use?
4. What are assembly directives?
5. What are syscalls?
6. What is the function of a makefile?
7. What are assembly aliases?
8. What tool is used to disassemble an executable program?

3 Dealing with Memory

Overview of the chapter

Chapter 3 focuses on how RISC-V assembly interacts with memory, introducing key concepts such as loading, storing, and addressing. It builds on previous chapters by explaining how data is accessed, moved, and manipulated in memory during program execution. Unless specified otherwise, the majority of the programs throughout the book were built and executed on 64-bit systems[34].

3.1 Big and little endian

In a **Little-Endian** system, the **least significant byte (LSB)**—the "little end"—is stored at the lowest memory address, the more significant bytes are stored at higher addresses.

Consider a 4-byte hexadecimal word with a value: 0x12345678.

- The “big” end is 12 (most significant).
- The "Little" end is 78 (least significant)

In a **Little-Endian** system, the **least significant byte (LSB)**—the "little end"—is stored at the smallest memory address (the first position). The more significant bytes follow at higher addresses.

Table 3- 1 Litte endian layout

Address	Value
Base	78 (LSB)
Base +1	56
Base+2	34
Base+3	12 (MSB)

Big endian by contrast would look like:

Table 3- 2 Big-endian layout

Address	Value
Base	12(MSB)
Base +1	34

[34] *See page 111 for discussion regarding 32-bit and 64-bit addition behavior.*

Base+2	56
Base+3	78 (LSB)

RISC-V can technically use bi-endian, but implementations are nearly always littles endian. See Listing 3- 2 for an example.

3.2 Load and Store instructions

Memory addresses are *loaded* from memory into registers and *stored* back from registers to memory. Operations are with respect to memory, so loading from memory to registers is a *read* operation and storing from registers is a *write* operation. The method by which memory addresses are derived is known as addressing modes, and there are several. The code fragments in this chapter will show how to communicate with memory and will also introduce various addressing modes.

Load and store instructions can access memory. Data is loaded from memory, acted on, and then stored back to memory. This is termed the load-store *architecture*.

3.2.1 LOAD Instructions (Memory → Registers)

3.2.1.1 Examining memory with GDB

GDB can be used to examine memory. The format of the command is `x/nfu addr`. The parameters have the following meaning:

Table 3- 3 Using GDB to display memory contents

n	# Amount to display
f	This is the display format; the default is to display in hex. The main options are o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char), s(string)
u	Unit size b = byte, h = halfword (2 bytes) w = word (4 bytes) g = giant (8 bytes)
addr	Address in memory to show

Example

```
(gdb) x/16w  0x4100e0
0x4100e0:     0x6c6c6548    0x00000a6f    0x00000000    0x00000000
0x4100f0:     0x0000002c    0x00000002    0x00080000    0x00000000
0x410100:     0x004000b0    0x00000000    0x00000028    0x00000000
0x410110:     0x00000000    0x00000000    0x00000000    0x00000000
```

To examine memory pointed to by a label (`mymemorylocation`) the following syntax can be used –

```
(gdb) x /16xw &mymemorylocation
0x11104:      0x0000abcd    0x00001234    0x00000000    0x00000000
0x11114:      0x36410000    0x72000000    0x76637369    0x002c0100
0x11124:      0x72050000    0x69343676    0x5f307032    0x3070326d
```

```
0x11134:      0x7032615f    0x32665f30    0x645f3070    0x5f307032
```

3.2.1.2 Load and Store example

Listing 3- 1 below shows a basic example of how to read from and write to memory.

The first instruction, `la, t0, word1` the address (here it is 0x11110) of word1 into register t0. The data is identified by the label `word1` in the `.data` section of the code. The contents of the address are loaded into the 64-bit register t1, since the instruction is load word and the upper 32-bits of the destination register are sign-extended, giving a 64-bit value of 0xffffffffabcd1234 (since bit 31 is a 1). The load word unsigned instruction treats the upper 32-bits differently; it pads them with zeros, giving a result of 0x00000000abcd1234.

The next instruction, `la t3, bufferspace` is the destination address for the data that will be loaded into memory. The address is identified by the label `bufferspace.`

Next, the instruction `sd t1, 0(t3)` stores the doubleword held in t1 into the memory address pointed to by register t3 (bufferspace), Finally, `sd t2,8(t3)` stores the 64-bits in register t2 into memory eight places (the offset) from the start of bufferspace. The format of store is to specify the source location into a memory address specified by a register added to an offset.

Listing 3-1 Basic read (load) and write (store) memory operation

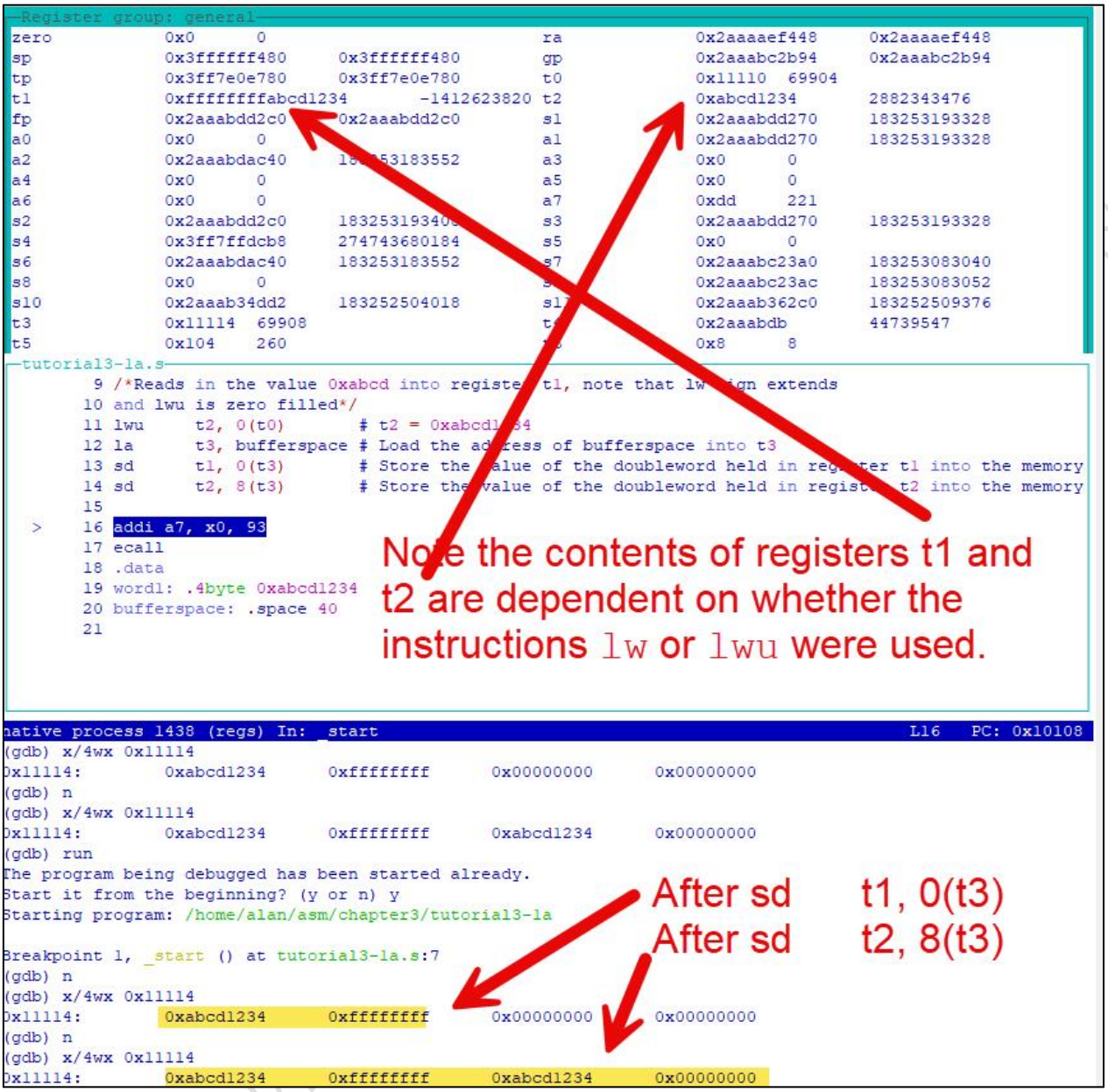
```
/*Listing 3 1 Basic read (load) and write (store) memory operation, the program defines four bytes, and copies them to a defined memory location (bufferspace), illustrating load and store operations*/

.section .text

.global _start

_start:

.option norelax

la      t0, word1

lw      t1, 0(t0)       # t1 = 0xffffffffabcd1234

/*Reads in the value 0xabcd into register t1, note that lw sign extends and lwu is zero filled*/

lwu     t2, 0(t0)       # t2 = 0xabcd1234

la      t3, bufferspace # Load the address of bufferspace into t3

sd       t1, 0(t3)        # Store the value of the doubleword held in register t1 into the memory location pointed to by register t3 plus an offset of 0.

sd       t2, 8(t3)        # Store the value of the doubleword held in register t2 into the memory location pointed to by register t3 plus an offset of 8.


addi a7, x0, 93

ecall

.data

word1: .4byte 0xabcd1234
```

3.2.1.3 GDB trace of listing3-1 showing the memory contents

```
bufferspace: .space 40

Breakpoint 1, _start () at listing3-3.s:6

6           la      t0, word1

(gdb) n

7           lw      t1, 0(t0)       # t1 = 0xffffffffabcd1234

(gdb)

9           lwu     t2, 0(t0)       # t2 = 0xabcd1234

(gdb)

10          la      t3, bufferspace # Load the address of bufferspace into t3

(gdb) i reg t0

t0             0x11110  69904

(gdb) i reg t1

t1             0xffffffffabcd1234       -1412623820

(gdb) i reg t2

t2             0xabcd1234       2882343476

(gdb) n

11           sd       t1, 0(t3)        # Store the value of the doubleword held in register t1 into
the memory location pointed to by register t3 plus an offset of 0.

(gdb) n

12           sd       t2, 8(t3)        # Store the value of the doubleword held in register t2 into
the memory location pointed to by register t3 plus an offset of 8.

(gdb) n

14      addi a7, x0, 93

(gdb) x /16wx &bufferspace

0x11114:        0xabcd1234      0xffffffff      0xabcd1234      0x00000000

0x11124:        0x00000000      0x00000000      0x00000000      0x00000000

0x11134:        0x00000000      0x00000000      0x00003641      0x73697200

0x11144:        0x01007663      0x0000002c      0x36767205      0x70326934
```

A graphical trace with GDB (see page 100 on how to set up the more enhanced version of GDB) is instructive.

Figure 3- 1 GDB trace of listing3-1



Note the `sd` command stores a double word (64bits at a time). Note: store word (`sw`) instruction's operand order.

It is important to understand how the `lw` and `lwu` instructions differ. Load word (LW) loads 32-bits into the lower half of the 64-bit register and sign-extends the upper 32-bits. Load word unsigned (`lwu`) zero-fills the upper 32-bits of the register.

From here on, GDB will be displayed using the **T**ext **U**ser **I**nterface!

The next program further clarifies how data is stored in little-endian format:

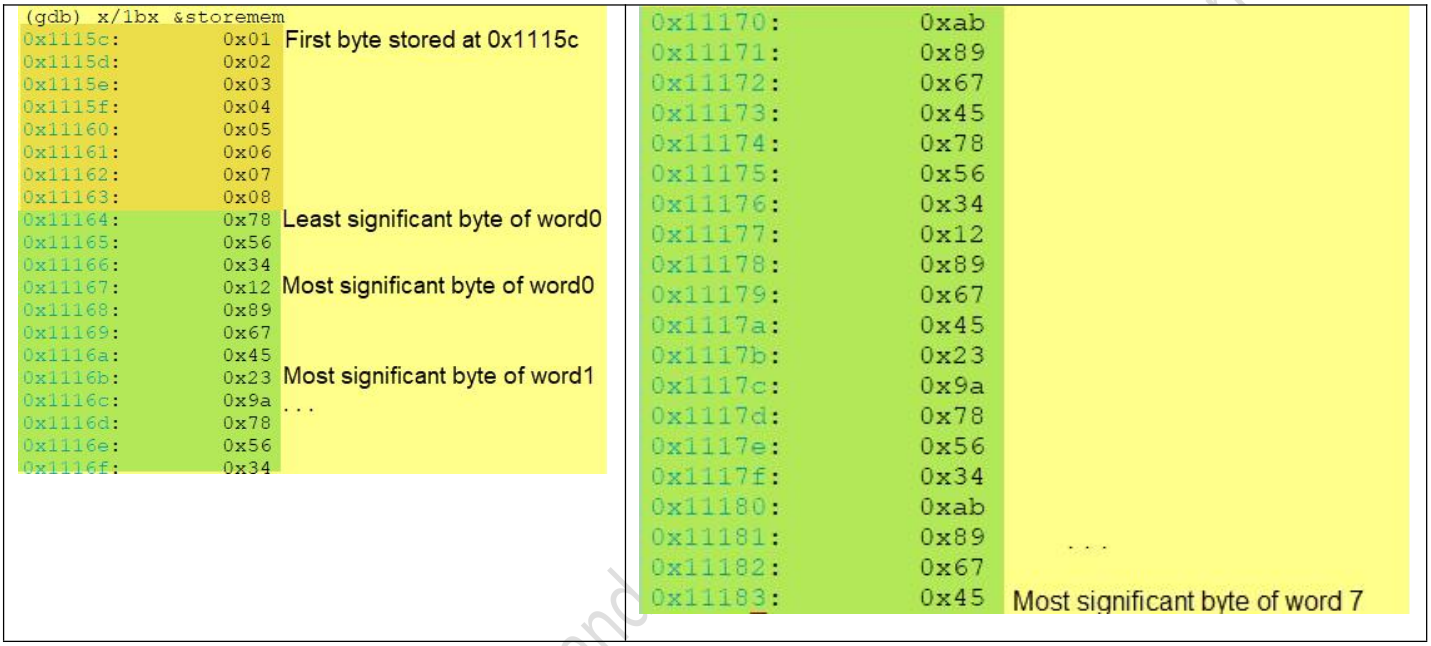
Listing 3-2 Illustrating little-endian storage

```
# Listing 3-2
```

```
.data
# Define the raw data
bytes: .byte 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08
words: .word 0x12345678, 0x23456789, 0x3456789a, 0x456789ab, 0x12345678, 0x23456789, 0x3456789a, 0x456789ab
storemem: .space 40
.text
.global _start


_start:
    la a0, bytes      # Source for bytes
    la a1, words      # Source for words
    la a2, storemem   # Base address for data storage


# Load and Store 8 Bytes ---
    li t2, 0              # Loop counter
byte_loop:
    lbu t0, 0(a0)         # Load 1 byte into t0
    sb  t0, 0(a2)         # Store 1 byte immediately to destination


    addi a0, a0, 1        # Increment source pointer
    addi a2, a2, 1        # Increment destination pointer
    addi t2, t2, 1        # Increment counter
    li   t3, 8
    blt  t2, t3, byte_loop


# Load and Store 8 Words ---
    li t2, 0              # Reset counter
word_loop:
    lw  t1, 0(a1)         # Load 1 word into t1
    sw  t1, 0(a2)         # Store 1 word immediately to destination


    addi a1, a1, 4        # Increment source by 4 (word size)
    addi a2, a2, 4        # Increment destination by 4
```

```
    addi t2, t2, 1        # Increment counter

    li   t3, 8

    blt  t2, t3, word_loop

# Quit

    li a7,93

    ecall
```

The memory layout looks like:



Note the byte values are stored sequentially, since only one byte is processed at a time, whereas the word value is four bytes wide and stored in order of the lowest (least significant) byte first.

3.3 Outputting (Writing) ASCII text

The next listing shows how text can be sent to the standard output device (stdout) – the screen. The *write* syscall will do this job and it has the decimal value of 64. Three registers (a0, a1, a2) hold the parameters required by this syscall and are set up as shown in Table 3- 2.

Table 3- 4Parameters required by the Write syscall

Register	Parameter meaning
a0	Holds value 1 (stdout)
a1	Hold the address of the output text (located at the label message)
a2	Contains the length of the output text (12 characters)

Note the message string is terminated by the *newline* character (/n)

Listing 3-3 Use of the Write Syscall

```
# Listing 3-3
.section .text
.global _start
_start:
li a0, 1 # use a0 for stdout
la a1, message # Load the address of the message text
li a2, 12 # Store the message length
li a7, 64    # Write syscall
ecall
li a7, 93  # Exit syscall
ecall
.data
message: .ascii "Hello RISCV\n"
```

Execute the program with the command –

```
./listing3-3
Hello RISCV
```

The unaliased version of this program is shown below:

```
objdump -d -M no-aliases listing3-2
listing3-3:     file format elf64-littleriscv
Disassembly of section .text:
00000000000100e8 <_start>:
100e8:       00100513           addi    a0,zero,1
100ec:       00001597   auipc   a1,0x1
100f0:       01c58593   addi    a1,a1,28 # 11108 <__DATA_BEGIN__>
100f4:       00c00613   addi    a2,zero,12
100f8:       04000893   addi    a7,zero,64
100fc:       00000073   ecall
10100:       05d00893   addi    a7,zero,93
10104:       00000073  ecall
```

GDB can be used to show the memory layout -

```
(gdb) x /16c &message
0x11108:   72 'H'  101 'e' 108 'l' 108 'l' 111 'o' 32 ' '  82 'R'  73 'I'
```

```
0x11110:  83 'S'  67 'C'  86 'V'  10 '\n' 65 'A'  54 '6'  0 '\000' 0 '\000'
```

This shows that the ASCII characters are laid out starting at the lowest address (0x11108) then counting upwards to 0x 11113.

3.4 Inputting (reading) values

The next example shows how to read a value using the read syscall. The read syscall uses the value 63 and places the input into memory defined by the symbol `buffer.`

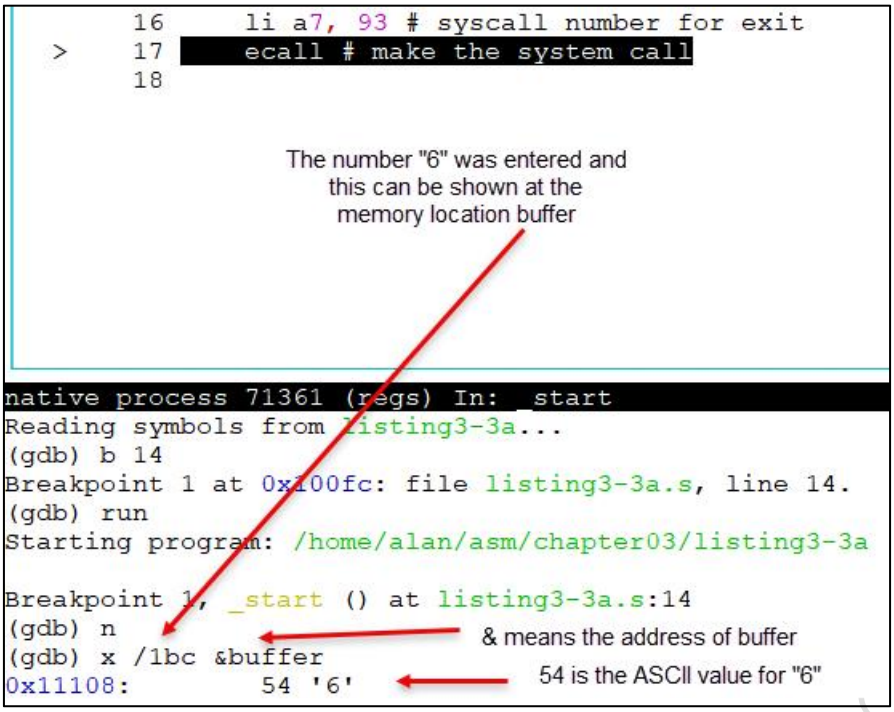
Table 3- 5 Parameters required by the read syscall

Register	Parameter meaning
a0	Holds the value 0 (stdin)
a1	Hold the address of the storage buffer
a2	Contains the length of the input characters

Listing 3-4Input operation

```
#Listing 3-4
# This is a simple program that reads in a single digit
.section .data
buffer:
.space 1
.section .text
.global _start
_start:
li a0, 0 # file descriptor 0 (stdin)
la a1, buffer # address of the buffer
li a2, 1 # number of bytes to read
li a7, 63 # Read syscall
ecall
li a7, 93 # syscall number for exit
ecall # make the system call
```

The GDB session below shows that the memory location buffer holds the value 54 decimal which is the ASCI code of the character “6”.



3.5 Relative and Absolute Addressing

Program Counter (PC) relative addressing is used to reference locations relative to the program counter. For example, a location could be accessed as PC +100 which would refer to a location 100 places beyond the current program counter's contents. Execution of consecutive (non-branch/jump) instructions advances the program counter by four, since instructions have a width of 32-bits (4 bytes). It is important to facilitate both forward and backward locations. Absolute addressing refers to the actual location in memory where an instruction or data resides.

3.5.1 RISC-V Assembler Modifiers

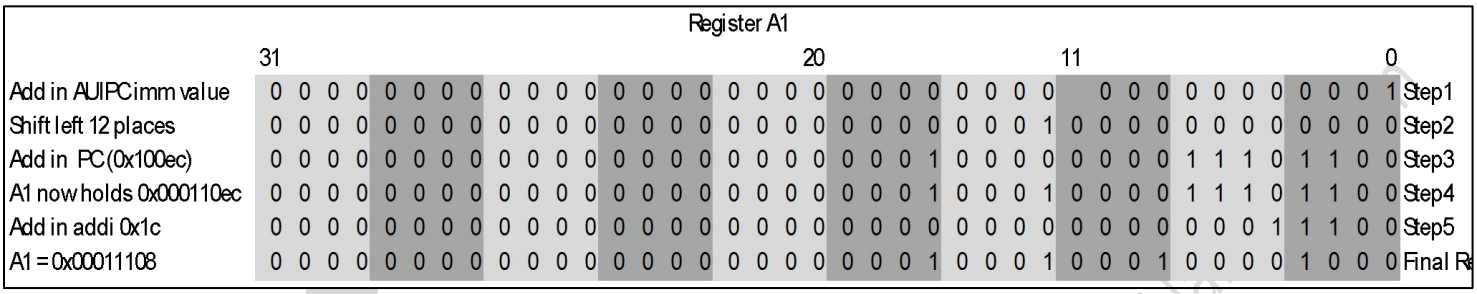
The assembler supports instructions to generate relative and absolute addresses. The address is broken up into the 12-bit lower portion (lo) and a 20-bit upper portion (hi). Before discussing this topic in detail - consider how the pseudo instruction *LA* breaks into the instructions *AUIPC* and *ADDI*:

```
la a1, message
```

Disassembles to →

```
00000000000100e8 <_start>:
100e8:       00100513
li      a0,1
100ec:       00001597            auipc   a1,0x1
100f0:       01c58593              addi    a1,a1,28 # 11108 <__DATA_BEGIN__>
```

Figure 3- 2auipc and addi instruction example to generate an address



The upper 20 bits (from `AUIPC`) are taken from the Program Counter's current contents (0x110ec) and the lower 12 bits (from `ADDI`) are 0x01c.

The immediate value of 0x1 is placed in the a1 register (from the `AUIPC` instruction) and is then shifted 12 places, causing it to occupy the upper 20 bits of the register. Register a1 now holds the value 0X00001000. This is added to the value in the program counter, giving 0x110ec. Next, the immediate value (0x1C) (from the `ADDI` instruction) is added to the contents of a1 and placed in register a1, so a1 now contains 0x11108.

Using the GDB command `info variables` shows:

```
All defined variables:

Non-debugging symbols:

0x0000000000011108  __DATA_BEGIN__

0x0000000000011108  message

0x000000000001111f  __SDATA_BEGIN__

0x000000000001111f  __bss_start

0x000000000001111f  _edata

0x0000000000011120  __BSS_END__

0x0000000000011120  _end
```

This confirms that the address of the string message resides at 0x11108. It also shows the value of the pseudo-instruction `LA` which is much easier to use. The assembler also helps us if we do not use the pseudo instructions. The instructions can resolve addresses by using modifiers such as `%lo, %hi, %pcrel_hi` and `%pcrel_lo`.

Table 3- 6 Absolute and relative adressing

Modifier	Format/Example	Description
%hi	`lui a1, %hi(symbol)`	Loads the upper 20 bits of the symbol's address into register a1
%lo	`addi a1, a1, %lo(symbol`	Loads the lower 12 bits of the symbol's address into register a1

%pcrel_hi	`auipc a2, %pcrel_hi(symbol)`	Loads the high 20 bits of a relative address between the PC and symbol
%pcrel_lo	`addi a2, a2, %pcrel_lo(label)`	Loads the high 20 bits of a relative address between the PC and label

The reason two instructions are needed is that there is no single instruction capable of loading a 32-bit immediate value. Referring back to the I-type and U-type instructions, there are instructions that load 12 bits and instructions that load 20 bits. Combining them is how a 32-bit immediate value is achieved.

- LUI is a U-type instruction and sets the low order bits to zero in the destination register and fills in the high order bits.
- ADDI is an I-type instruction and adds the low-order bits to the destination register.
- AUIPC sets the destination register's high-order bits to the sum of the immediate value and the program counter with the lo order bits set to zero.

The next listing shows an example of PC-relative addressing.

Listing 3-5 Relative addressing example

```
# Listing 3-3
# This listing shows how to use PC-Relative addressing using modifiers
.section .data
message:
.ascii "This is a line of text\n"
.equ writecall, 64
.equ exitcall, 93
.equ stout, 1
.equ stringlength, 23
.section .text
.global _start
_start:
li a0, stout # stdout
label1: auipc a1,%pcrel_hi(message) # Loads upper 20 bits
addi a1,a1,%pcrel_lo(label1) # Loads lower 12 bits
li a2, stringlength # String length
li a7, writecall # Write syscall
ecall
li a7, exitcall # syscall number for exit
ecall # make the system call
```

Error! Bookmark not defined.he next piece of code shows how absolute addressing is achieved with `%lo` and `%hi`.

Listing 3-6 Using absolute addressing with %lo and %hi

```
#Listing 3-6
#This listing shows how to generate absolute addressing using %lo and %hi modifiers
.section .data
message:
.ascii "This is a line of text\n"
.equ writecall, 64
.equ exitcall, 93
.equ stout, 1
.equ stringlength, 23
.section .text
.global _start
_start:
li a0, stout # stdout
lui a1, %hi(message) # Loads upper 20 bits of mesages' absolute address
addi a1,a1,%lo(message) # Loads lower 12 bits of message's absolute address
li a2, stringlength # String length
li a7, writecall # Write syscall
ecall
li a7, exit call # syscall number for exit
ecall # make the system call
```

The first absolute addressing instruction `lui ai, %hi(message)` loads a1 with the value 0x11000, the next instruction `addi a1, a1, %lo(message)` adds 0x108 to a1 giving a final result of 0x11108 which is the absolute address of the symbol *message*.

3.6 Linker Relaxation

You may wonder why code such as that written below does not run correctly.

Listing 3-7 Linker relaxation codeexample

```
.section .data
        msg1: .ascii "This is the first string\n"
        msg2: .ascii "This is the second string\n"
        .equ msg1len, 25
        .equ msg2len, 26
```

```
        .section .text
        .global _start
_start:
        li a0, 1
        la a1,msg1
        li a2, msg1len
        li a7, 64
        ecall

        li a0, 1
        la a1,msg2
        li a2, msg2len
        li a7,64
        ecall
exit:
        li a7,93
        ecall
```

Output:

```
./print2strings
This is the first string
$
```

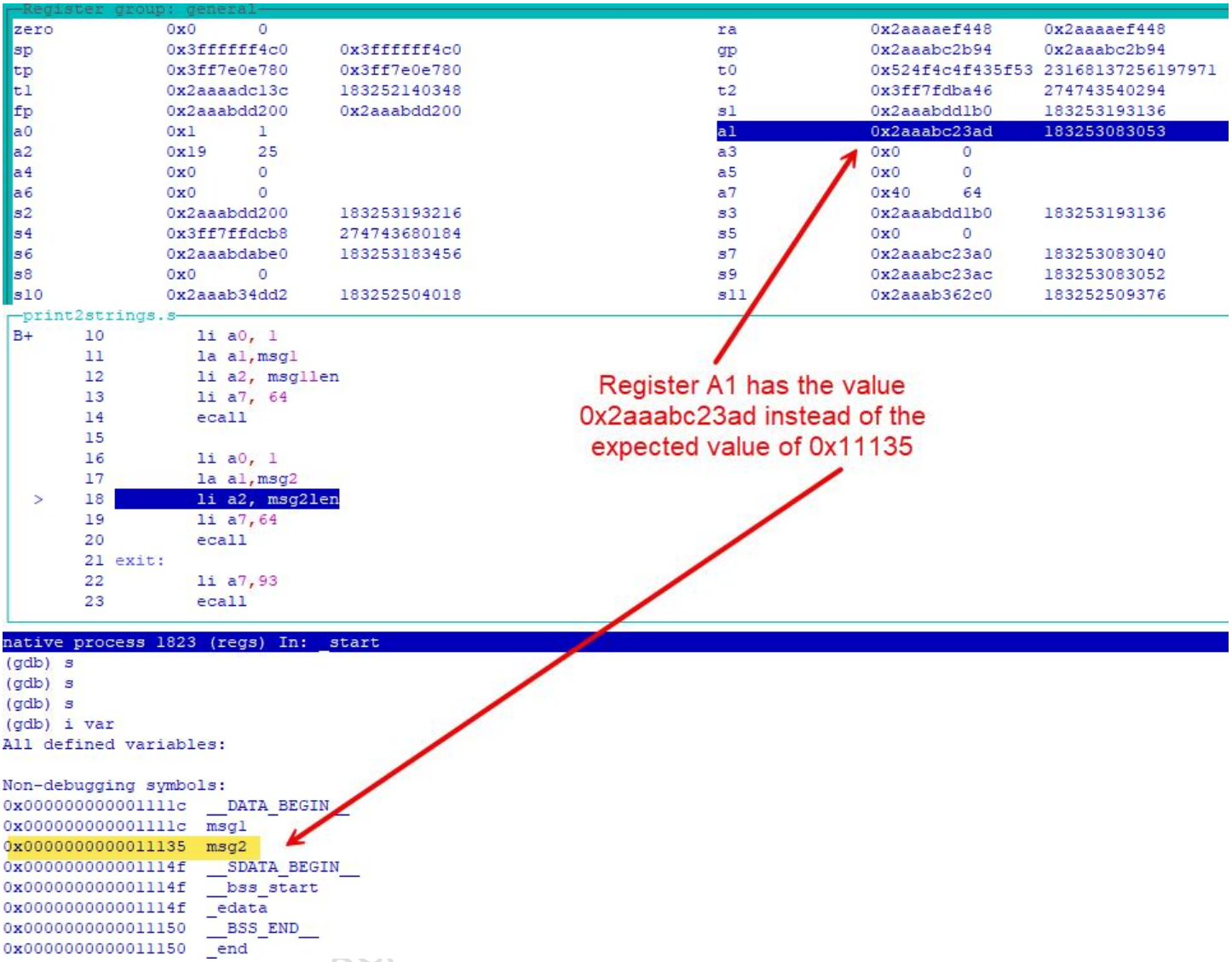
The program only printed the first string!

Using GDB to trace the program shows that after the instruction `la a1,msg2` has executed, register a1 has a value of `0x2aaabc23ad`.

This is not what we expected??

Running the command `info variables` in GDB shows that the base address of `msg2` is `0x11135`.

Figure 3- 3 GDB illustrating non initialized GP register



The reason this happened is that the pseudo-instruction `LA` is trying to translate it into a single instruction. We have seen that the load address (`LA`) uses the upper 20 bits (`AUIPC`) are taken from the Program Counter's current contents) and the lower 12 bits from `ADDI`. Now if the variables span across a small address range, it is possible to specify the address by adjusting the 12-bit offset with the addi instruction and keeping the upper 20 bits constant. In this case, the load address instruction will only need the `ADDI` part of the pseudo-instruction.

If we use objdump, we can see that the first load into register a1 uses the two instructions located at `0x100ec` and `0x100f0` but the second load into register a1 at address `0x10104` only uses the single `ADDI` instruction and skips `AUIPC`.

```
000000000000100e8 <_start>:

   100e8:       00100513                li      a0,1

   100ec:       00001597                auipc   a1,0x1

   100f0:       03058593                addi    a1,a1,48 # 1111c <__DATA_BEGIN__>
```

```
   100f4:       01900613                li      a2,25
   100f8:       04000893                li      a7,64
   100fc:       00000073                ecall
   10100:       00100513                li      a0,1
   10104:       81918593                addi    a1,gp,-2023 # 11135 <msg2>
   10108:       01a00613                li      a2,26
   1010c:       04000893                li      a7,64
   10110:       00000073                ecall
0000000000010114 <exit>:
   10114:       05d00893                li      a7,93
   10118:       00000073                ecall
```

We could avoid the pseudo-instruction la and always *manually* use `auipc` and `addi` to refer to the full 32-bit address; however, we can avoid this and force the `LA` instruction to *automatically* use the two instructions `AUIPC` and `ADDI` with the directive `.option norelax` as shown below.

Listing 3-8 print two strings

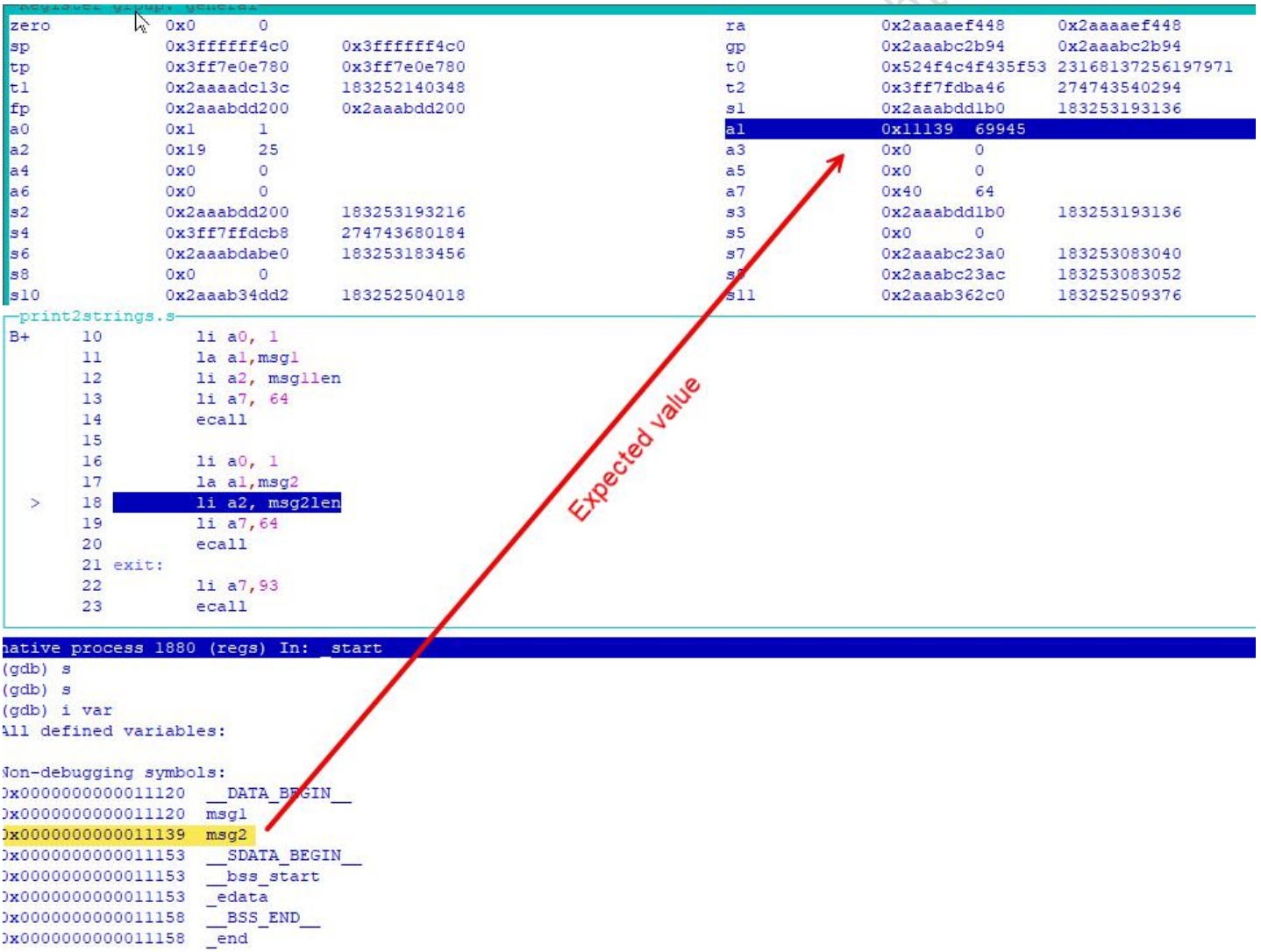
```
# Listing 3-8
.section .data
        msg1: .ascii "This is the first string\n"
        msg2: .ascii "This is the second string\n"
        .equ msg1len, 25
        .equ msg2len, 26
        .section .text
        .global _start
_start:
        .option norelax
        li a0, 1
        la a1,msg1
        li a2, msg1len
        li a7, 64
        ecall

        li a0, 1
        la a1,msg2
        li a2, msg2len
```

```
        li a7,64

        ecall

exit:

        li a7,93

        ecall
```

The program now runs correctly –

```
This is the first string

This is the second string

$
```

Running the code through GDB shows:



The `objdump` utility now shows:

```
objdump -d print2strings
```

```
print2strings:     file format elf64-littleriscv
Disassembly of section .text:
00000000000100e8 <_start>:
   100e8:       00100513                li      a0,1
   100ec:       00001597                auipc   a1,0x1
   100f0:       03458593                addi    a1,a1,52 # 11120 <__DATA_BEGIN__>
   100f4:       01900613                li      a2,25
   100f8:       04000893                li      a7,64
   100fc:       00000073                ecall
   10100:       00100513                li      a0,1
   10104:       00001597                auipc   a1,0x1
   10108:       03558593                addi    a1,a1,53 # 11139 <msg2>
   1010c:       01a00613                li      a2,26
   10110:       04000893                li      a7,64
   10114:       00000073                ecall
```

So in summary:

- The directive `.option norelax` is used to disable *linker relaxation*.
- Relaxation is used to optimize performance by reducing the number of instructions when the program's address range is limited.

Consider the next two listings.

Listing 3-9 Non-relaxed version of code

```
# Listing 3-9
# This version does not use relaxation
.section .data
ask:
.ascii "Please input a character\n"
.align 2
confirm:
.ascii "You entered: \n "
.align 2
linefeed:
.ascii "\n"
```

```
buffer:
.space 4
.section .text
.global _start
_start:
.option push # Save context
.option norelax # Turn off relaxation to set up the global pointer
1:auipc gp, %pcrel_hi(__global_pointer$)
addi  gp, gp, %pcrel_lo(1b)# b for back
.option pop # Now restore relaxation
.option norelax
li a0, 1 #stdout
la a1, ask #Text for first output string
li a2, 27 #String length
li a7, 64 # Write syscall
ecall
li a0, 0 # file descriptor 0 (stdin)
la a1, buffer # address of the buffer
li a2, 1 # number of bytes to read
li a7, 63 # Read syscall
ecall


li a0, 1 #stdout again
la a1, confirm # Text for second output string
li a2, 15#Length
li a7, 64 #Write syscall
ecall
li a0, 1 #stdout again
la a1, buffer
li a2, 1 #Length
li a7, 64 #Write syscall
ecall
```

```
# Tidy up with a newline!
li a0, 1
la a1, linefeed
li a2, 1
li a7, 64
ecall
li a7, 93 # syscall number for exit
ecall # make the system call
```

The numeric label 1 is suffixed with ‘b’ or ‘f’ for backward and forward references respectively.

Listing 3-10 Relaxed version of code

```
# Listing 3-10
# This version uses relaxation
.section .data
ask:
.ascii "Please input a character\n"
.align 2
confirm:
.ascii "You entered: \n "
.align 2
linefeed:
.ascii "\n"
buffer:
.space 4
.section .text
.global _start
_start:
.option push # Save context
.option norelax # Turn off relaxation to set up the global pointer
1: auipc gp, %pcrel_hi(__global_pointer$)
addi  gp, gp, %pcrel_lo(1b)
.option pop # Now restore relaxation state
# .option norelax is commented out in this version
li a0, 1 #stdout
```

```
la a1, ask #Text for first output string
li a2, 27 #String length
li a7, 64 # Write syscall
ecall
li a0, 0 # file descriptor 0 (stdin)
la a1, buffer # address of the buffer
li a2, 1 # number of bytes to read
li a7, 63 # Read syscall
ecall
li a0, 1 #stdout again
la a1, confirm # Text for second output string
li a2, 15#Length
li a7, 64 #Write syscall
ecall
li a0, 1 #stdout again
la a1, buffer
li a2, 1 #Length
li a7, 64 #Write syscall
ecall
# Tidy up with a newline!
li a0, 1
la a1, linefeed
li a2, 1
li a7, 64
ecall
li a7, 93 # syscall number for exit
ecall # make the system call
```

As we have seen, linker relaxation is used to provide more efficient coding. It is not always necessary to specify the full 32-bit address range, as many sections of code can run in the 12-bit range (minus 2048 to plus 2047 bytes) without having to use `AUIPC` to load the upper 20-bits.

There are several types of linker relaxation; however, only *global pointer relaxation* will be discussed here.

The *global pointer* can be used to specify an offset. So, rather than having to specify two instructions, we can drop `auipc` and use only one instruction with the global pointer as an offset.

This is an optimization performed by the linker, as it has a global view of all the files that will be linked together. Table 3- 7[35] shows how relaxation reduces the code size and enhances performance. Since the contents of the `.data` section are small enough to fit into 12 bits, the upper 20 bits need not be fetched each time.

The real gain is not so much the size of the code but its performance. Code that uses repetitive loop iterations can benefit greatly in terms of a reduction in execution time.[36]

Table 3- 7 Comparison of relaxed and non-relaxed code

Listing 3- 9 Non relaxed version of code

```
00000000000100e8 <_start>:
100e8:     00002197      auipc   gp,0x2
100ec: 88818193  addi  gp,gp,-1912 # 11970 <__global_pointer$>
100f0:     00100513      addi    a0,zero,1
100f4:     00001597      auipc   a1,0x1
100f8:  07c58593  addi    a1,a1,124 # 11170 <__DATA_BEGIN__>
100fc:     01b00613      addi    a2,zero,27
10100:     04000893      addi    a7,zero,64
10104:     00000073      ecall
10108:     00000513      addi    a0,zero,0
1010c:     00001597      auipc   a1,0x1
10110:     09158593      addi    a1,a1,145 # 1119d <buffer>
10114:     00100613      addi    a2,zero,1
10118:     03f00893      addi    a7,zero,63
1011c:     00000073      ecall
10120:     00100513      addi    a0,zero,1
10124:     00001597      auipc   a1,0x1
10128:     06858593      addi    a1,a1,104 # 1118c <confirm>
1012c:     00f00613      addi    a2, zero,15
10130:     04000893      addi    a7, zero,64
10134:     00000073      ecall
10138:     00100513      addi    a0, zero,1
1013c:     00001597      auipc   a1,0x1
10140:     06158593      addi    a1,a1,97 # 1119d <buffer>
```

Listing 3- 10 Relaxed version of code

```
00000000000100e8 <_start>:
100e8:  00002197      auipc   gp,0x2
100ec:  87818193   addi    gp,gp,-1928 # 11960 <__global_pointer$>
100f0:  00100513      addi    a0,zero,1
100f4:  00001597  auipc   a1,0x1
100f8:  06c58593      addi    a1,a1,108 # 11160 <__DATA_BEGIN__>
100fc:      01b00613  addi    a2,zero,27
10100:      04000893 addi    a7,zero,64
10104:      00000073  ecall
10108:      00000513  addi    a0,zero,0
1010c:     82d18593 addi    a1,gp,-2003 # 1118d <buffer>
10110:      00100613  addi    a2,zero,1
10114:      03f00893    addi    a7, zero,63
10118:      00000073   ecall
1011c:      00100513    addi    a0, zero,1
10120:      81c18593  addi    a1, gp,-2020 # 1117c <confirm>
10124:      00f00613     addi    a2, zero,15
10128:      04000893     addi    a7, zero,64
1012c:      00000073     ecall
10130:      00100513     addi    a0, zero,1
10134:      82d18593   addi    a1,gp,-2003 # 1118d <buffer>
10138:      00100613     addi    a2, zero,1
1013c:      04000893     addi    a7, zero,64
10140:      00000073     ecall
```

[35] *The listings were generated by* `objdump -d -M no-aliases <file>`

[36] *Refer to RISC-V ABIs Specification (https://lists.riscv.org/g/tech-psabi/attachment/61/0/riscv-abi.pdf) section 8.5.5 for more information on the global offset table.*

```
10144:    00100613      addi   a2,zero,1
10148:    04000893      addi   a7,zero,64
1014c:    00000073      ecall
10150:    00100513      addi   a0,zero,1
10154:    00001597      auipc  a1,0x1
10158:    04858593      addi   a1, a1,72 # 1119c <linefeed>
1015c:    00100613      addi   a2, zero,1
10160:    04000893      addi   a7, zero,64
10164:    00000073      ecall
10168:    05d00893      addi   a7, zero,93
1016c:    00000073      ecall
```

```
10144:    00100513   addi   a0, zero,1
10148:    82c18593 addi   a1,gp,-2004 # 1118c <linefeed>
1014c:    00100613   addi   a2, zero,1
10150:    04000893   addi   a7, zero,64
10154:    00000073   ecall
10158:    05d00893   addi   a7, zero,93
1015c:    00000073   ecall
```

auipc count (non-relaxed)

```
100e8:        00002197                auipc   gp,0x2
100f4:        00001597                auipc   a1,0x1
1010c:        00001597                auipc   a1,0x1
10124:        00001597                auipc   a1,0x1
1013c:        00001597                auipc   a1,0x1
10154:        00001597                auipc   a1,0x1
```

auipc count (relaxed)

```
100e8:        00002197                auipc   gp,0x2
100f4:        00001597                auipc   a1,0x1
```

3.6.1 Further relaxation example

The global pointer is set up as an offset in the middle of the 12-bit address space and uses the linker-defined symbol `__global_pointer$` for initialization. The listing below disables linker relaxation, initializes the GP register, and then re-enables relaxation.

The intent of global relaxation is to convert the instructions –

`auipc + addi → single addi` when possible.

Listing 3-11 Further example of linker relaxation use

```
# Listing 3-11 Basic read (load) and write (store) memory operation.

# The program defines four bytes, and copies them to a defined memory location # (bufferspace),
illustrating load and store operations.

# This version uses linker relaxation, thus saving an instruction using the

# gp register.
```

```
.section .text
.global _start
_start:
  .option relax
  .option push # Save the options state
  .option norelax # Turn off relaxation to get the global Pointer value
  la gp, __global_pointer$
  .option pop # Restore the options state, with relaxation enabled
  la  t0, word1
  lw      t1, 0(t0)   # t1 = 0xffffffffabcd1234
  /*Reads in the value 0xabcd into register t1, note that lw sign extends and lwu is zero filled*/
  lwu t2, 0(t0)       # t2 = 0xabcd1234
  la  t3, bufferspace # Load the address of bufferspace into t3
  sd        t1, 0(t3)    # Store the value of the doubleword held in register t1 into the memory
location pointed to by register t3 plus an offset of 0.
  sd  t2, 8(t3) # Store the value of the doubleword held in register t2 into the memory location
pointed to by register t3 plus an offset of 8.
  addi a7, x0, 93
  ecall
.data
  word1: .4byte 0xabcd1234
  bufferspace: .space 40
```

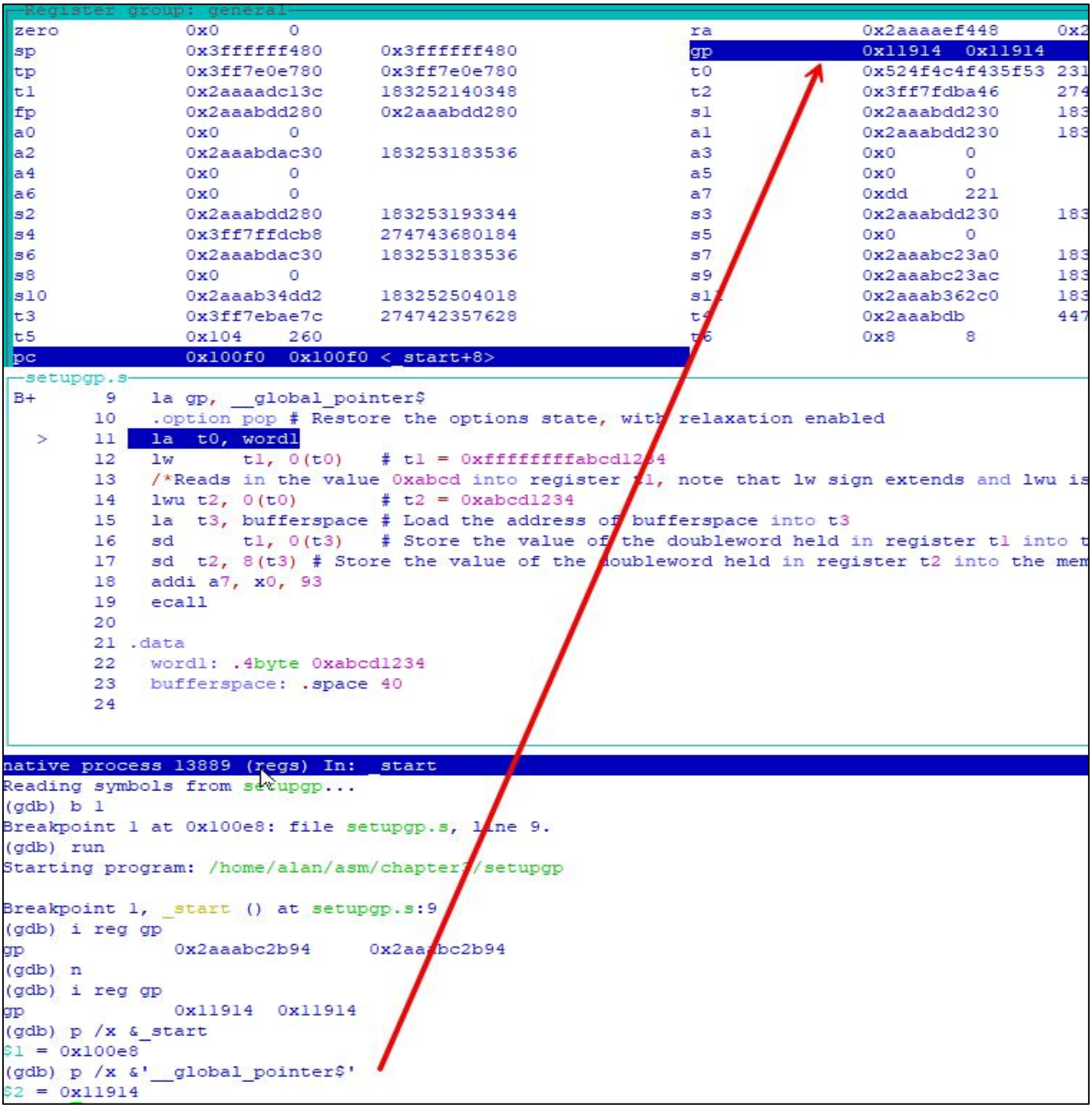
The disassembly from objdump is shown next –

```
00000000000100e8 <_start>:
100e8: 00002197 auipc gp,0x2
100ec: 82c18193 addi  gp,gp,-2004 # 11914 <__global_pointer$>
100f0: 00001297 auipc  t0,0x1
100f4: 02428293 addi   t0,t0,36 # 11114 <__DATA_BEGIN__>
100f8: 0002a303        lw      t1,0(t0)
100fc: 0002e383        lwu     t2,0(t0)
10100: 80418e13        addi    t3,gp,-2044 # 11118 <bufferspace>
10104: 006e3023        sd      t1,0(t3)
10108: 007e3423        sd      t2,8(t3)
1010c: 05d00893        li      a7,93
```

```
10110: 00000073        ecall
```

A GDB trace is shown below:

Figure 3- 4 Using GDB to show initialized GP register



The assembler can be modified to generate non-relaxed code with the `-mno-relax` option. To modify the make file to include it edit the `makefile` to read –

```
TARGETFILE = $(targetfile)

print: $(TARGETFILE).o

ld -o $(TARGETFILE) $(TARGETFILE).o

$(TARGETFILE).o: $(TARGETFILE).s

as -mno-relax -g -o $(TARGETFILE).o $(TARGETFILE).s
```

For the remaining programs in the book, the `makefiles`[37] (unless stated otherwise) will include the `-mno-relax` option.

3.6.2 Enhancements to GDB

GDB can be used in default mode for analyzing code. Entering the following commands into the file `~/.gdbinit` will give a better (TUI) layout experience.

```
layout split

layout regs

set history save on

set history filename ~/gdbhistory

set logging enabled on
```

Note that if using the GDB TUI, then the up and down arrows are no longer available for command history; use Ctrl-P(revious) and Ctrl-N(ext) instead.

Another great tool is GDBGUI.

Installation instructions can be found at www.gdbgui.com.

With Debian, it can also be installed with `pipx` using `pipx install gdbgui`, followed by `pipx ensurepath`. If `pipx` is not installed, then install it with `sudo apt install -y pipx`.

Start `gdbgui` from the command line by entering the following command:

```
gdbgui --args <executable program>
```

The screenshot is a snapshot of the program midway through. The memory location shows the values in hex. The GDB command window is at the bottom left. A subset of the registers is shown along with the source code.

[37] *For reasons already discussed linker relaxation can be helpful in performance-oriented applications and then a different makefile would be used.*

Figure 3-5 GDBGUI

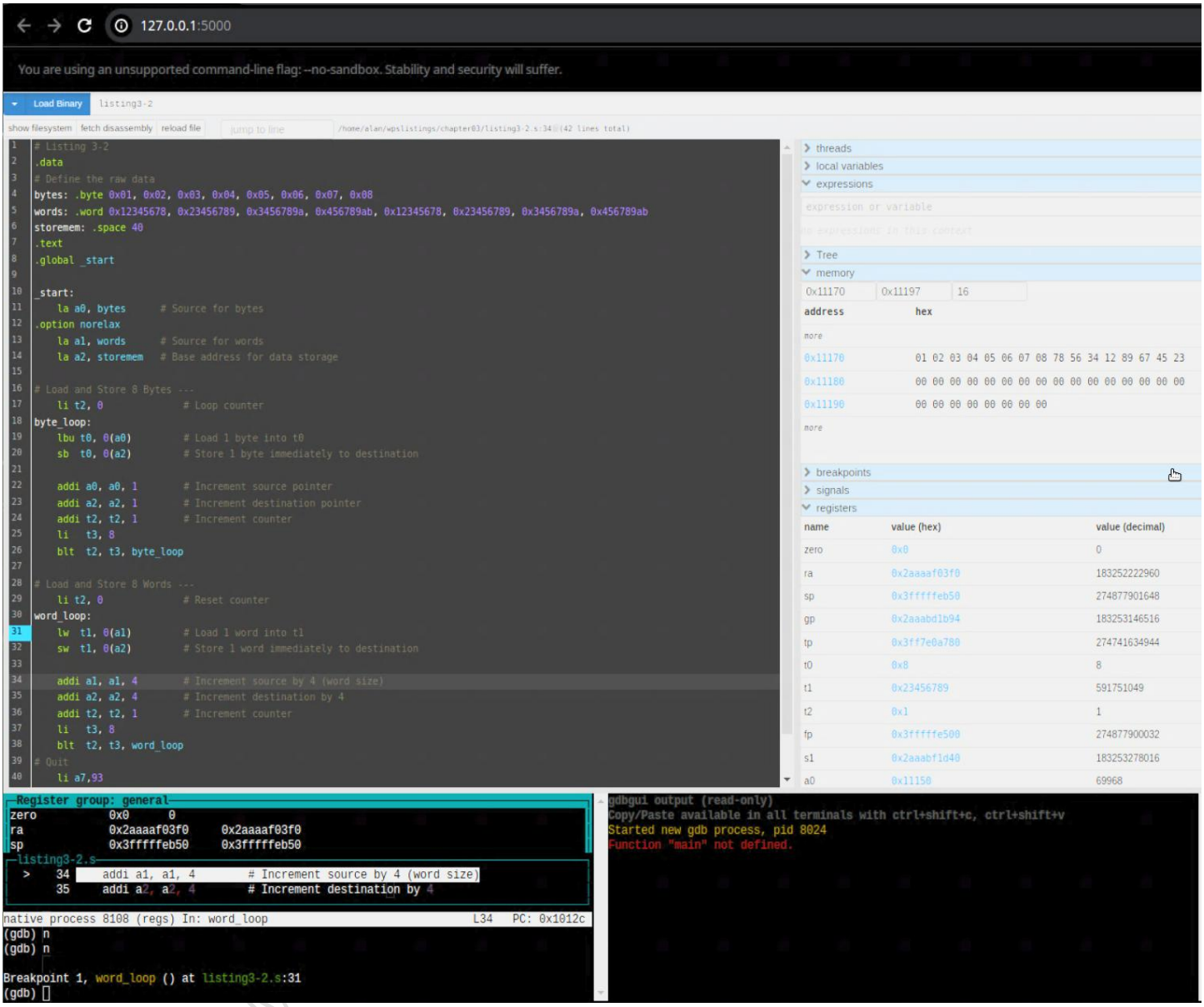
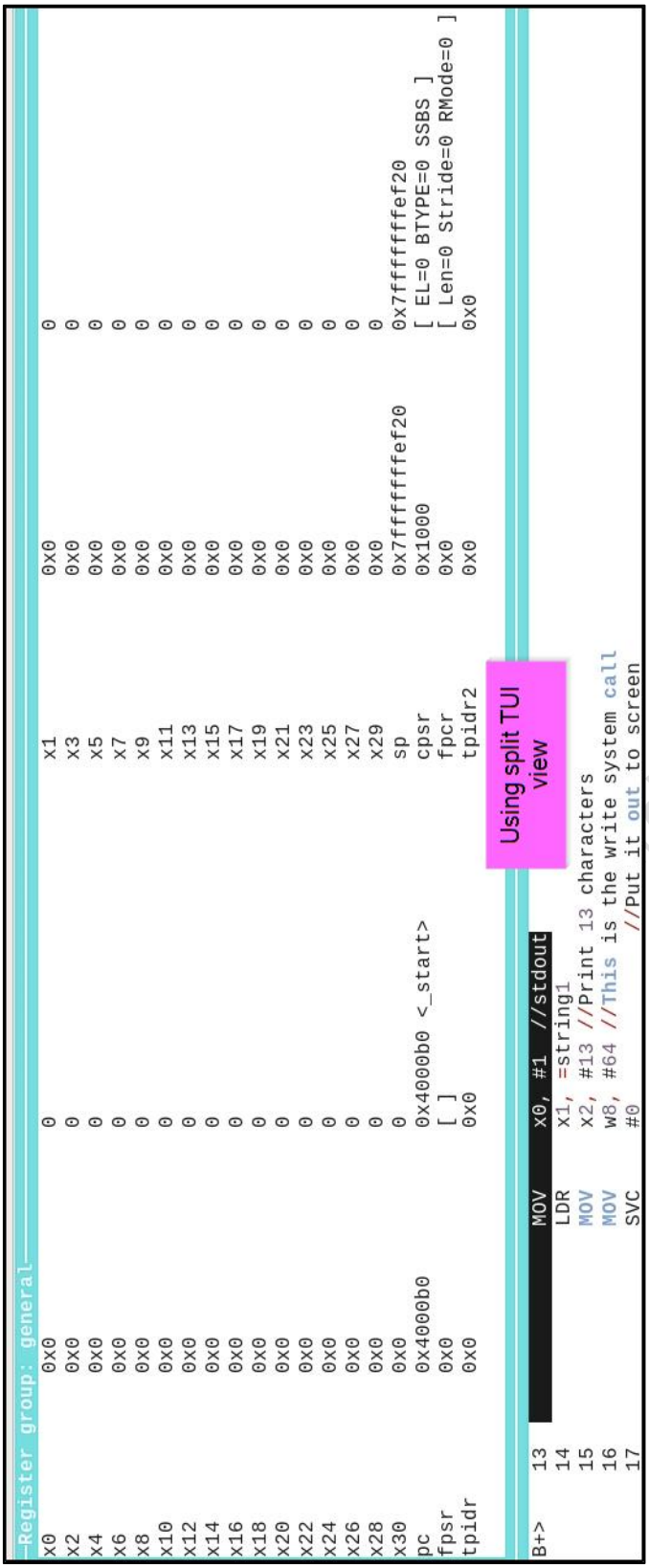


Figure 3- 6 GDB using TUI



3.7 Exercises for chapter3

1. Write a program that takes a user-inputted string, printing out hexadecimal codes for each character in the string; for example –

   “This is the input string”

   character Hex value

   T 54

   h 68

   . . .

2. Describe the purpose of linker relaxation.

3.7.1 RISC-V instructions covered in chapter 3

Load Instructions:

LB – Load byte

LBU – Load byte unsigned

LH – Load halfword

LHU – Load halfword unsigned

LW – Load word

LWU – Load word unsigned (64-bit systems)

LD – Load doubleword

Store Instructions:

SB – Store byte

SH – Store halfword

SW – Store word

SD – Store doubleword

System Call and Immediate Instructions:

LI – Load immediate (pseudo-instruction)

LA – Load address (pseudo-instruction)

ECALL – Environment call (used for invoking syscalls)

Addressing & Assembler-Related:

AUIPC– Add upper immediate to PC (used in PC-relative addressing)

Assembler modifiers like %lo(symbol) and %hi(symbol) are also discussed to support **absolute addressing**.

4 Arithmetic Operations (First Pass)

Overview of the chapter

Chapter 4 focuses on arithmetic and logical operations in RISC-V assembly. It builds upon the memory-handling concepts of Chapter 3 by introducing how to perform calculations, data manipulation, and conditional logic using registers. Floating-point operations are deferred, as there is a dedicated chapter for this topic.

4.1 Data Sizes

RISC_V uses the data sizes listed in Table 4- 1.

Table 4- 1Data Types

# of bits	Definition
8	Byte
16	Halfword
32	Word
64	Doubleword

Load and Store instructions designate variants of these data sizes with the following abbreviations:

- W: Word
- H: Halfword
- HU: Halfword unsigned
- B: Byte
- BU: Byte unsigned

4.2 Integer Instructions

4.2.1 Register ADD

The first listing shows the ADD instruction, which has the format `add rd`(estination), `rs`(ource)`1`, `rs`(ource)`2`. In this case, the addition of source register t0 and source register t1 is sent to the destination register t3.

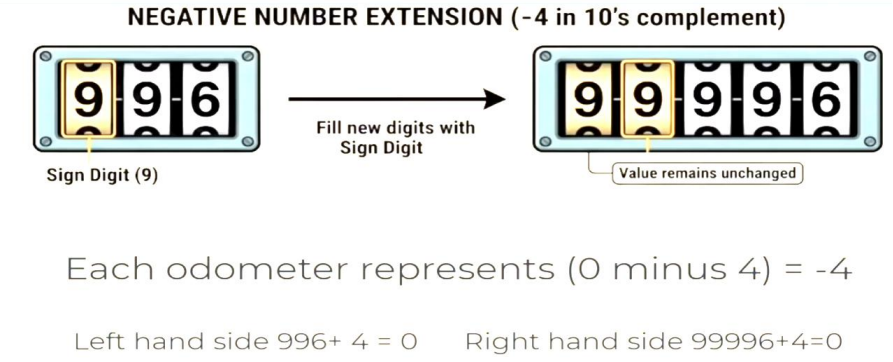
The instruction addw is an example of a 64-bit instruction operating on 32-bit values, which gives a 64-bit result of 0xffffffffffdc9999. The result is *sign-extended* by propagating the sign bit to preserve the sign. If the value was positive, then the result would be 0x00000000ffdc9999 or simply 0xffdc9999.

Sign extension is used when converting signed smaller numbers to signed larger values. Essentially the upper part of the larger number is padded with the sign bit (bit 31 when using addw).

The following analogy may help with the concept of sign extension by showing how a three-digit counter is extended to a higher-resolution five-digit counter without losing its value.

Both counters represent the value minus four.

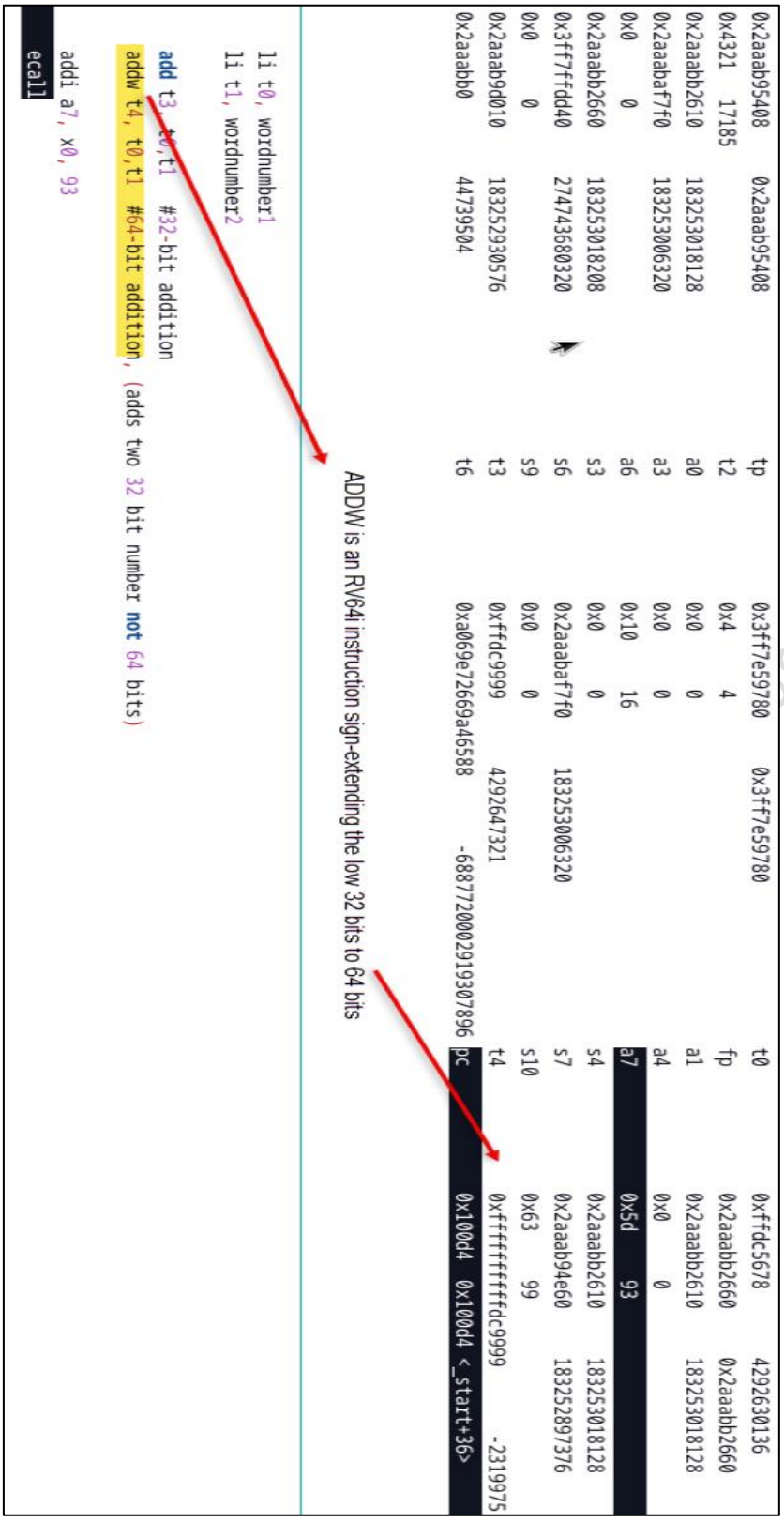
Figure 4-1 Sign-extension analogy



Listing 4-1 ADD and ADDW instructions

```
/* Listing 4-1 Simple addition (register)instruction
64-bit (add) and 32-bit (addw) are shown */
.section .data
.equ wordnumber1, 0xffdc5678 # four digit hex value
.equ wordnumber2, 0x4321
.equ wordnumber3, 0xfffdc5678 # nine digit hex value
.section .text
.global _start
_start:
  li t0, wordnumber1
  li t1, wordnumber2
  add t3,t0,t1  #64-bit addition # t3=0xffdc9999
  addw t4,t0,t1 #32-bit addition #t4=0xffffffffffdc9999
  li t0, wordnumber3
  add t3,t0,t1  #64-bit addition t3=0xfffdc9999
  addw t4,t0,t1 #32-bit addition t4=0xffffffffffdc9999
# no difference in addw as upper 32 bits are not used
  li a7, 93
  ecall
```

Figure 4-2 ADD and addw instructions



The instruction ADD gives a 64-bit result of 0xffdc9999

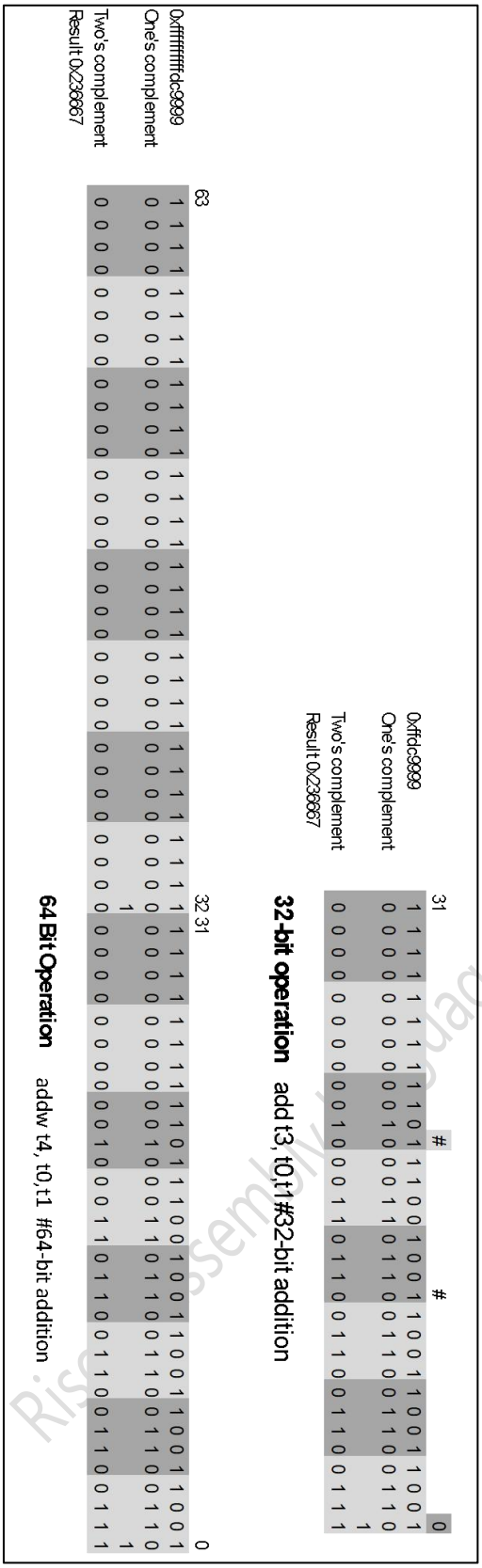
The instruction ADDW operates on the low-order 32 bits of each register and sign-extends to give a 64-bit result.

Note the difference between ADD and addw

- ADDW takes bits 0-31 of each register and gives a sign-extended 64-bit result.
- It is not valid for RV32.
- ADD takes bits 0-63 of each register and generates a 64-bit result.

The figure below shows that sign-extending a 32-bit result ensures that the results are consistent.

Table 4-2 Sign extension example



The disassembly for the .text section is as follows:

```
Disassembly of section .text:


00000000000100b0 <_start>:
   100b0:       001002b7                lui     t0,0x100
   100b4:       dc52829b                addiw   t0,t0,-571 # ffdc5 <__global_pointer$+0xee4d5>
   100b8:       00c29293                slli    t0,t0,0xc
   100bc:       67828293                addi    t0,t0,1656
   100c0:       00004337                lui     t1,0x4
   100c4:       3213031b                addiw   t1,t1,801 # 4321 <wordnumber2>
   100c8:       00628e33                add     t3,t0,t1
   100cc:       00628ebb                addw    t4,t0,t1
   100d0:       010002b7                lui     t0,0x1000
   100d4:       dc52829b                addiw   t0,t0,-571 # fffdc5 <__global_pointer$+0xfee4d5>
   100d8:       00c29293                slli    t0,t0,0xc
   100dc:       67828293                addi    t0,t0,1656
   100e0:       00628e33                add     t3,t0,t1
   100e4:       00628ebb                addw    t4,t0,t1
   100e8:       05d00893                li      a7,93
   100ec:       00000073                ecall
```

The `li t0, 0xffdc5678` instruction breaks down into:

```
lui t0, 0x100
lddiw t0, t0, -571
slli t0, t0, 0xc
addi t0, t0, 1656
```

This results in t0 being equal to 0XFFDC5678 as shown in Figure 4- 2.

Figure 4-3 Calculating LI to, 0xffdc5678 non-aliased steps

T0	31												20												11								0
LUI t0, 0x1 0x100000	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADDIW t0, 0xffdc5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0	0	1	0	1		
SLLI t0, t0, 0xffdc5000	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0		
ADDI 1656 0xffdc5678	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0		

The instruction `SLLI` has not been met before. This instruction performs a left shift by the number of places in the immediate operand, which means shifting the value currently in t0 12 places to the left.

4.2.2 ADD Immediate

The ADDI (ADD immediate instruction) has the form `addi rd, rs, imm12`. The source register is added to a 12-bit immediate value, and the result is placed in the destination register. The format of this instruction has already been described.

The addi instruction is straightforward –

Listing 4-2 ADDI example

```
/* Listing 4-2 Simple addition (immediate) instruction */
.section .data
        .equ wordnumber1, 0xffdc5678
        .equ wordnumber2, 0x87654321
        .equ myfirstconstant, 0x1ff
        .equ mysecondconstant, 0x321
.section .text
.global _start
_start:
  li t0, wordnumber1
  li t1, wordnumber2
  addi t3, t0,myfirstconstant #t3=0xffdc5877
  addiw t4, t1,mysecondconstant#t4= 0xffffffffff7654642
  li a7,  93
  ecall
```

The `addi` and `addiw` instructions are shown below.

Figure 4-4 Illustrating the ADD and ADDIW instructions

4.2.2.1 RV32 vs RV64 addi behavior

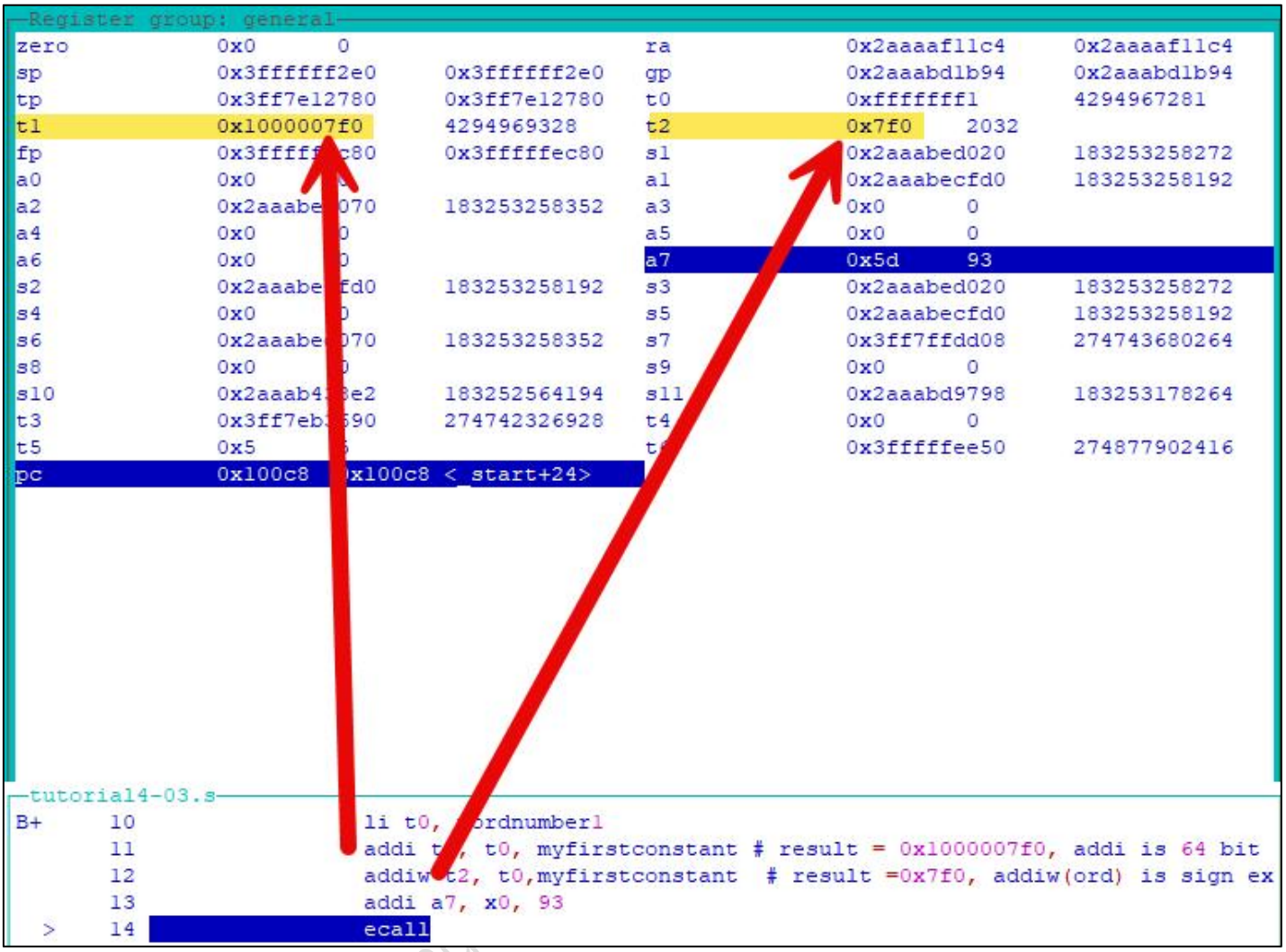
- The ADDI instruction uses the full native XLEN architecture.

- The ADDIW instruction is not valid on RV32 systems and adds the low-order 32 bits of rs to the 12-bit immediate field, then sign-extends the result to rd.

The `ADDIW` instruction adds the sign-extended immediate value to rs1 and then writes the result, which is sign-extended to rd as shown in the snippet below.

```
/*  Sign extension with addiw, note how addi is different on 32-bit and 64-bit systems.
Compare addi on a 64-bit to addiw on a 64-bit system */
.section .data
        .equ wordnumber1, 0xffffffff1
        .equ myfirstconstant, 0x7ff
.section .text
.global _start
_start:
    li t0, wordnumber1
    addi t1, t0, myfirstconstant # result = 0x1000007f0, addi is 64 bit native
     addiw t2, t0,myfirstconstant  # result =0x7f0, addiw(ord) is sign extended
      addi a7, x0, 93
      ecall
```

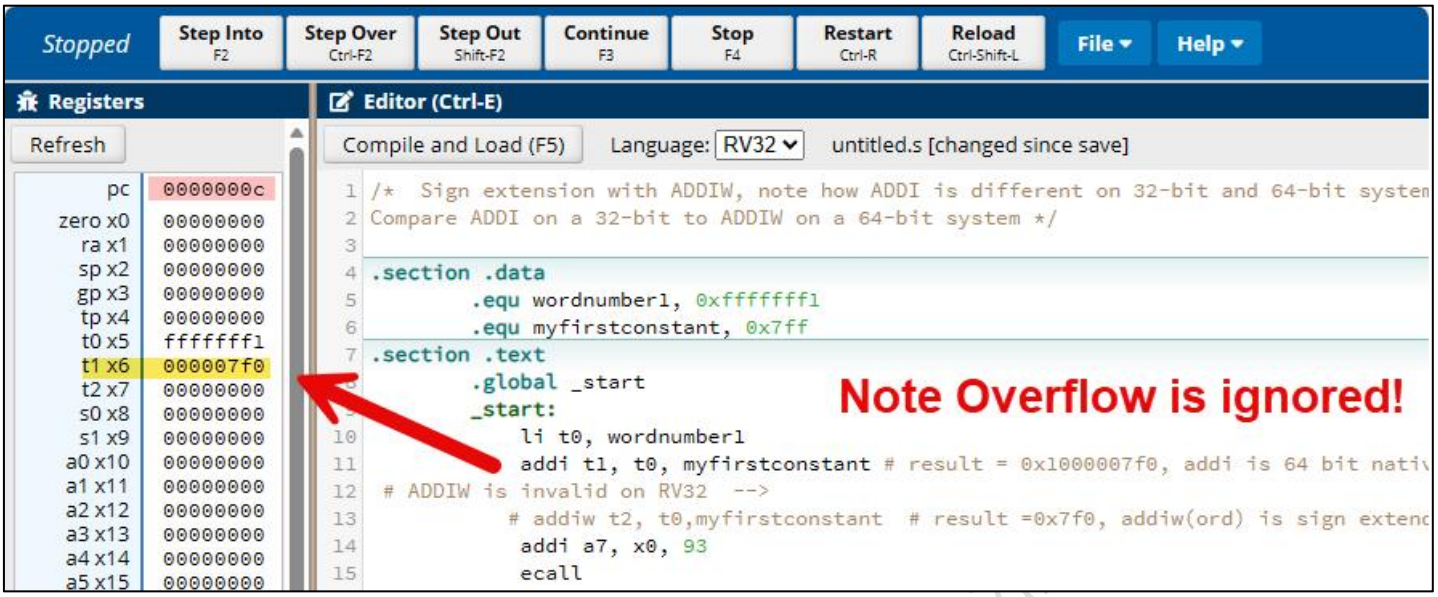
Figure 4-5 GDB trace comparing ADD (64-bit) with addiw (64-bit)



Now compare the result of the `addi` instruction running on a 32-bit system to the `addiw` result obtained in the previous program.

Running on RV32 with CPUlator shows:

The instruction `ADDIW` is not valid for 32-bit since W(ord) is the default data width. In this instance, the addition has caused a negative number to go positive. Note this is not flagged! Overflow with the same operands on a 64-bit system did not occur.



Finally, the next addition example shows another pitfall –

Listing 4-3 Sign extension

```
/*  Sign extension with addiw, note how addi is different on 32-bit and 64-bit systems.

Compare addi on a 32-bit to addiw on a 64-bit system */

.section .data

        .equ wordnumber1, 0xf00000001

        .equ myfirstconstant, 0xf

.section .text

.global _start

_start:

   li t0, wordnumber1

    addi t1, t0, myfirstconstant # result = 0x1000007f0, addi is 64 bit native


/* Using addiw with the same parameters gives 0x10; the 64-bit value 0xf00000001 was truncated to
the 32-bit

value of 0x00000001. The truncated value and the constant 0xf were added together, placing the
result in register

t2*/

        addiw t2, t0,myfirstconstant  # result = 0x10

        addi a7, x0, 93

        ecall
```

Note. In many cases, the numerical values encountered during day-to-day coding easily fit into the XLEN registers without any issues. Rather than check for overflow conditions after each arithmetic operation, it may be opportune to only check if there are reasons to believe that the number bounds may have been exceeded. This was historically more of an issue for 8-bit systems.

4.2.3 MV instruction

The MV instruction is aliased to addi. The format is `mv rd, rs` as shown in. After execution, the contents of t0 will have been copied[38] to t1.

Listing 4-4 MV instructionrun

```
/* Listing 4-4

Move instruction, actually a pseudo instruction

mv rd, rs --> addi rd, rs, 0*/

.section .data

.equ number1, 0x12345678

.section .text

.global _start

_start:

li t0, number1

mv t1, t0

addi a7, x0, 93

ecall
```

The unaliased listing is

```
-<_start>:

100b0: 123452b7            lui    t0,0x12345

100b4: 6782829b            addiw  t0,t0,1656 # 12345678 <number1>

100b8: 00028313            addi   t1,t0,0

100bc: 05d00893            addi   a7,zero,93

100c0: 00000073            ecall
```

4.2.4 SUB instruction

The available subtraction instructions are SUB and SUBW, there is no subtract-immediate variant, since this can be achieved through addition by adding a negative number.

Listing 4-5 Use of sub and subw instructions

```
# Listing 4-5
```

[38] *The Move instruction is really a copy function, in that the source register's contents are preserved.*

```
# Subtraction operations 32-bit (subw) and 64-bit (sub) are shown
.section .data
.equ wordnumber1, 0xffdc5678
.equ wordnumber2, 0x4321
.equ negativenumber, -4
.section .text
.global _start
_start:
li t0, wordnumber1
li t1, wordnumber2
sub t3, t0, t1  # 0x00000000ffdc1357;  positive result
sub t4, t1, t0  # 0xffffffff0023eca9; negative result
subw t5, t0, t1 # 0xffffffffffdc1357; Sign extended negative result, invalid for RV32
addi t2, t1, negativenumber #0x431d; subtracts 4 from t1 result --> t2
addi a7, x0, 93
ecall
```

Note the results obtained by using SUB and SUBW with the same operands.

4.3 Condition Codes

Many processors incorporate a condition code register (CCR) or status register to detect conditions such as

- Negative (N) True when a signed number is negative; false if positive.
- Zero (Z) True if the result, such as the comparison of values, is equal; false if not equal.
- Carry (C) True if a carry or no-borrow condition occurs, shifted-out bit.
- Overflow (V) True if an overflow condition occurs.

This is important for processors that have limited register sizes. Checking for these conditions takes time and for many programs, conditions such as overflow and carry will never occur. This is the case where there is a finite number of elements, well below the maximum register data width size. A 32-bit register can hold over 4 billion positive integers, which will not be exceeded when dealing with real-world objects such as inventory, personnel, and weather temperatures, etc. Clearly, it would be wasteful to check for additive carry conditions when new personnel are hired. There are, however, situations where these conditions can occur, and in the case of RISC-V, this can be checked.

4.3.1 Detecting an oVerflow condition

An example of an *overflow* condition occurs when the data is too large to fit into a register. Consider a small eight-bit register that can hold *signed* values ranging from -128 to +127. Adding two positive numbers, such as 0x50 and 0x40 results in 90 which is a negative number in signed eight-bit arithmetic.

Table 4- 3 Detecting an overflow condition (signed)

50_{16} =	0 (Sign bit+)	1	0	1	0	0	0	0
40_{16} =	0 (Sign bit+)	1	0	0	0	0	0	0
+ = 90	1 (Sign bit-)	0	0	1	0	0	0	0

For unsigned, the result of an addition should not be a number smaller than either of the operands.

Table 4- 4 Detecting an overflow condition (unsigned)

73_{16} =	0	1	1	1	0	0	1	1
95_{16} =	1	0	0	1	0	1	0	1
+ = 90	0	0	0	0	1	0	0	0

Note the 9th bit has been discarded (fallen into the *bit bucket*)!

In general, for addition -

- For signed arithmetic, when the operands have the same sign but the result is a different sign, then overflow has occurred.
- Overflow does not occur if the numbers have different signs.
- For unsigned arithmetic, the addition should not be smaller than either of the operands.

Other conditions, such as Negative, Carry, and Borrows, can also be checked for by software rather than implementing dedicated registers.

4.3.2 RVM Instructions

The ADD and SUB instructions are part of the *Base Integer Set* (RVI). Multiply and Divide instructions belong to the optional *Multiply/Divide instruction set (RVM).*

4.3.3 Multiply Instructions

The MUL instruction has the format `mul rd, rs1, rs2.`

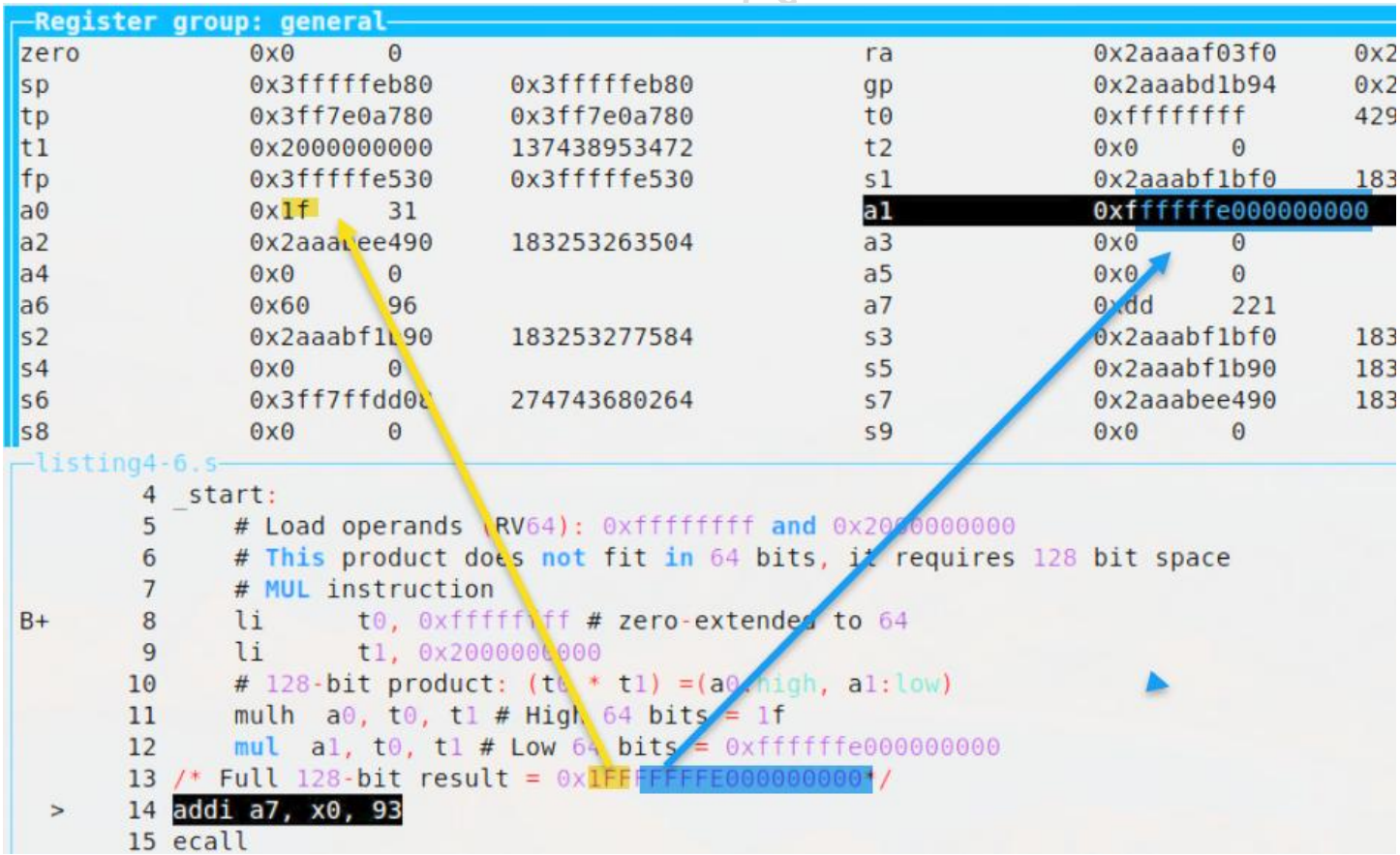
Listing 4-6 64-bit multiplication

```
/*Listing 4-6 This system ran on RV64M*/
    .text
    .globl _start
```

```
_start:
    # Load operands (RV64): 0xffffffff and 0x2000000000
    # This product does not fit in 64 bits, it requires 128 bit space
    # MUL instruction
    li      t0, 0xffffffff # zero-extended to 64
    li      t1, 0x2000000000
    # 128-bit product: (t0 * t1) =(a0:high, a1:low)
    mulh  a0, t0, t1 # High 64 bits = 1f
    mul  a1, t0, t1 # Low 64 bits = 0xfffffe000000000
/* Full 128-bit result = 0x1FFFFFFE000000000*/
addi a7, x0, 93
ecall
```

A GDB trace -

Figure 4-6 GDB trace of 128-bit multiplication



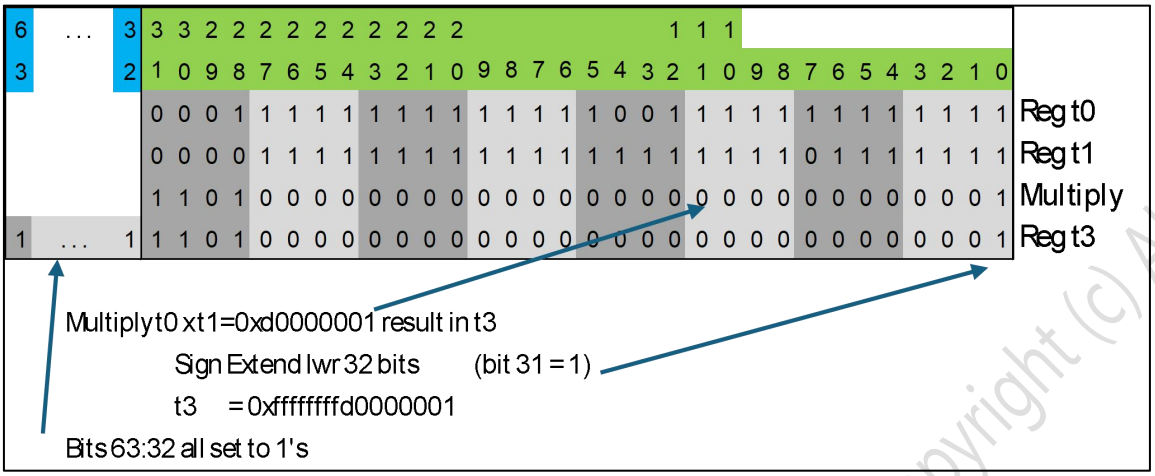
Listing 4-7 Further Multiply instructions on RV64

```
 /* Listing 4-7 Multiplication operations
This system ran on RV64M*/
.section .data
.section .text
.global _start
_start:
li t0, 0xffffffff
li t1, 0x2000000000
li a0, 0x7ff
li a1, 0x600
#RISC-V documents state to execute in the order of MULH, MULHU, MULHSU first then MUL
/*64-bit x 64 bit multiplication example giving a 128-bit result
Reg t2 holds bits 63-0
Reg t3 holds bits 127-64*/
mulh t3, t0, t1 # t3 = 0x1f Upper 64 bits *127-64)
mul t2, t0,t1   # t2 = 0xffffffe00000000 lower 64 bits (63:0)
# Overall 128 bit result is 0x1f fffffe00000000
mulh a2, a0, a1 # a2 = 0x0
mul a3, a0, a1 # a3 =  0x2ffa00
# Overall 128 bit result is 0x0x2ffa00
# Unsigned multiply
mulhu a2, a0, a1 # a2 = 0x00000000
mul a3, a0, a1 # a3 =   0x2ffa00
# Overall 64 bit result is 0x2ffa00

# First operand is signed, second operand is unsigned
mulhsu a2, t2, t2 # a2 = 0xffffffe000000400
addi a7, x0, 93
ecall
```

MULH is used to get the upper half of the product and, in conjunction with MUL can generate an XLENx2 value. So with RV64, two registers can be used to form a 128-bit result, which is held in two registers: one register holding bits 63:0 and the other holding bits 127:64.

The instruction MULW is a 64-bit instruction, multiplying the lower 32 bits of the source registers and sign extending bit 31 as shown in Figure 4- 6. Combining the upper 32 bits from MUL and the lower from MULW gives the result 0x01ffffffd0000001

Figure 4-7 MULW instruction



In this case, the product was sign-extended. The next instruction, where t0 has the value 0xfff and t1 has the value 0x8 gives a result of 0x7fff8 since the upper 32 bits from the MUL instruction equaled 0x0 and the lower 32 bits from the MULW instruction equaled 0x0007fff8.

4.3.4 Illustrating the mechanics of 64-bit multiplication going to 128 bits

Evaluating `mulh t3,t0,t1 -`

Assume t0 = 0xFFFFFFFF and t1=0x200000000

Step1 The instruction `mulh` writes bits 127-64 of the 64-bit x 64-bit product to rd, so `mulh t3,t0,t1` will multiply t0 by t1 placing bits 127-64 into register t3.

Step 2 The instruction `mul` handles the operands as signed 64-bit. It computes the 128-bit product and writes the low 64 bits of the product to rd, so `mul t2, t0,t1` places bits 63-0 of the 128-bit product into t2.

Step 3 Combine the two 64-bit registers together to show the 128-bit result →

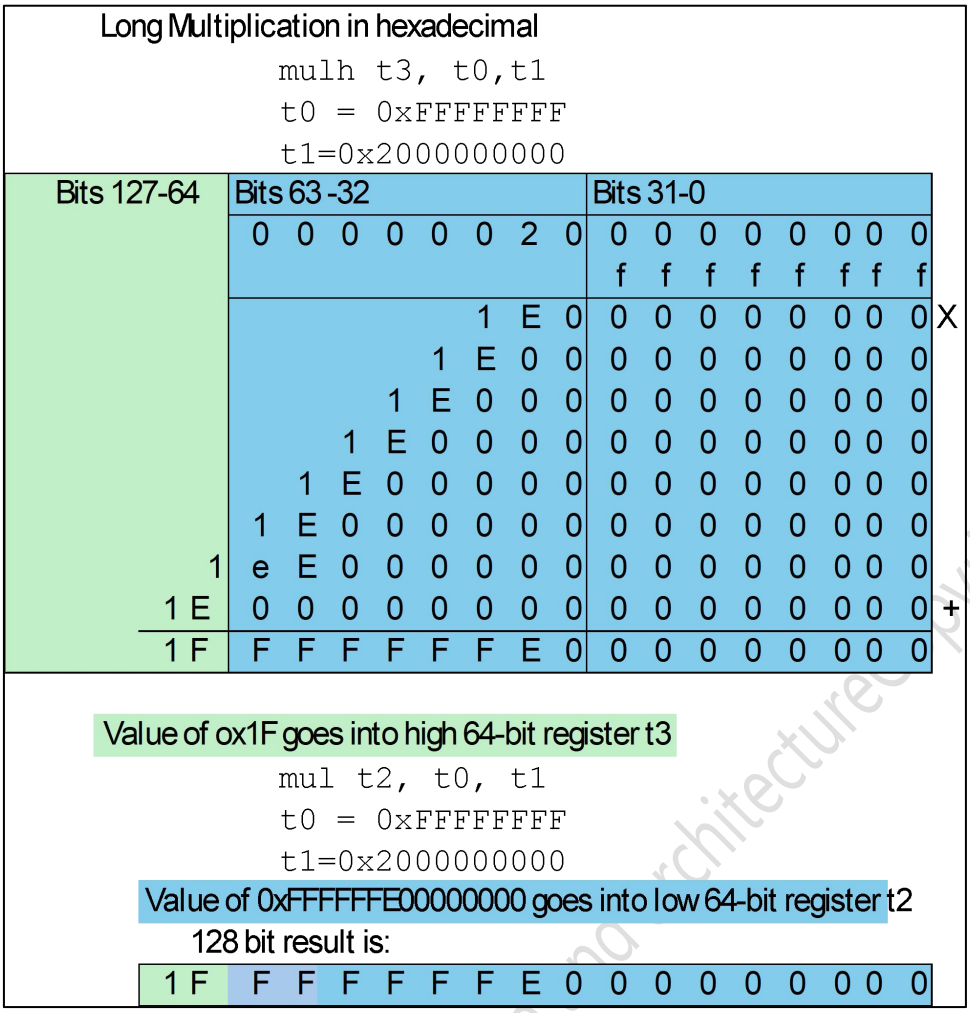
t3 = 0x1F

t2 = 0xFFFFFFE000000000

t3,t2= 0x1FFFFFFE000000000

A non-computer method using long multiplication is shown in Figure 4- 7

Figure 4-8 Using a manual long multiplication method to multiply two 64-bit hex numbers



To summarize multiplication –

Table 4-5 Summary of RVM Multiply Instructions

Instruction	Description	Data Polarity
MUL	Multiplies rs1 by rs2, result in rd, ignores overflow	
MULH	Multiplies the signed values of rs1 by rs2, upper half of the product goes into rd	Both operands are signed
MULHU	Multiplies the unsigned values of rs1 by rs2, upper half of the product goes into rd	Both operands are unsigned
MULHSU	Multiplies the signed value of rs1 by the unsigned value of rs2, upper half of the product going into rd	First operand is signed, the second is unsigned.

MULW	Lower 32-bits of the product,sign-extended.	Sign-extends to 64-bits.

4.3.5 Divide Instructions

Division is simpler than multiplication. The instructions are DIV(W) (Divide Signed) , UDIV(W) (Divide Unsigned), REM (Remainder Signed) and REMU (Remainder unsigned). A basic example follows –

Listing 4-8 Division example

```
/*Listing 4-8 Division operations*/
.section .data
# Define two 32-bit words
word1: .word 1025
word2: .word 4
.section .text
.global _start
_start:
       .option norelax
       lw t0, word1
       lw t1, word2
# Operand1 is the numerator
# Operand2 is the denominator
       div t2, t0,t1    # t2 = 0x100
       divu t3, t0,t1   # t3 = 0x100
       rem t4, t0,t1    # t4 = 0x1
       remu t5, t0, t1 # t5 = 0x1
       addi a7, x0, 93
       ecall
```

There are wide variants - DIVW and DIVUW are 64-bit instructions. These instructions divide the lower 32 bits of operand1 by the lower 32 bits of operand2. DIVW is for signed numbers, and DIVUW is for unsigned. The result is sign-extended. The remainder instruction counterparts are REMW and REMUW, also sign-extending to 64 bits.

4.3.5.1 *Division by zero*

Division by zero will generate an all 1's result (all bits are set) and is not trapped. The remainder is equal to the dividend. A further example is shown below:

Listing 4-9 Further Division examples

```
/*Listing 4-9 Further division operations*/
.section .data
```

```
# Define two bytes and a 32-bit word
word1: .word 0xffffffff1
byte1: .byte 1
byte2: .byte 4
.section .text
.global _start
_start:
lw t0, word1
lb t1, byte1
lb t2, byte2
mv t6, zero  # Use for division by zero
# Operand1 is the numerator
# Operand2 is the denominator
divw t3, t0, t1   # t3 = 0xfffffffffffffff1
remw t5, t0, t1   # t5 = 0x0
divuw t4, t0, t2  # t4 = 0x3ffffffc
remuw a0, t0, t2  # a0 = 0x1
# Divide by zero
div a1, t0, t6 # a1 = 0xffffffffffffffff (all ones)
rem a2, t0, t6 # a2 = 0xfffffffffffffff1
divw a3, t0, t6 # a3 = 0xffffffffffffffff
remw a4, t0, t6 # a4 = 0xffffffffffffff1
addi a7, x0, 93
ecall
```

Since division by zero gives the combination of all ones and the original dividend, it can be checked after the division has taken place when necessary. The order given of DIV followed by REM in the listing is recommended for microarchitecture efficiency.

4.4 Shift Operations

RISC-V offers several shift instructions. Left shifts can be register or immediate. Shift right instructions are similar except that they also offer a shift right arithmetic variant. This is summarized in Table 4- 6.

Table 4-6 RV32 Shift Instructions

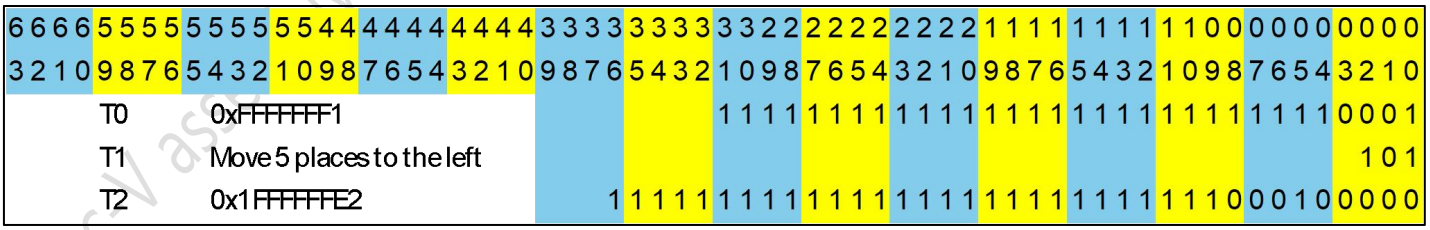
Instruction	Description	Syntax	Example
SLI	Shift Left Logical, shifts rs1 bits leftwards by count in rs2 fills moved empty bit positions with zeros, result in rd.	`sll rd, rs1, rs2`	`sll t2, t0, t1`
SLLI	Shift Left Logical Immediate, shifts rs1 bits leftwards by count in the immediate field, fills moved empty bit positions with zeros	`slli rd, rs1, imm`	`slli t3, t0, 18`
SRL	Shift Right Logical, shifts rs1 bits rightwards by count in rs2 fills moved empty bit positions with zeros, result in rd	`srl rd, rs1, rs2`	`srl t4, t0, t1`
SRLI	Shift Right Logical Immediate, shifts rs1 bits rightwards by count in the immediate field, fills moved empty bit positions with zeros, result in rd	`srli rd, rs1, imm`	`srli t5, t0, 10`
SRA	Shift Right Arithmetic, shifts rs1 bits rightwards by count in rs2 fills moved empty bit positions with the value of rs1's most significant bit, result in rd	`sra rd, rs1, rs2`	`sra t6, t2, t1`
SRAI	Shift Right Arithmetic Immediate, shifts rs1 bits rightwards by count in the immediate field, fills moved empty bit positions with the value of rs1's most significant bit, , result in rd	`srai rd, rs1, imm`	`srai t6,t2, 18`

RV64 features wide variants of the shift instruction which sign-extend to 64-bits – `slliw, srliw` and `sraiw.`

RV32 uses the five least significant bits (4:0) for the shift amount and RV64 will use the six least significant bits (5:0)[39]. Figure 4- 8 shows a five-bit shift to the left; note that bits 4:0 are replaced by incoming zeros. As discussed earlier each move to the left is equivalent to multiplying by two. In this example, the value 0xfffffff1 has been multiplied by 32 (2^5).

The figure shows how the instruction `sll t2, t0, t1` is handled. Here t1=5 and t0 =0xFFFFFFF1.

Figure 4-9 SLL instruction sll t2, t0, t1



Listing 4-10 Shift instructions

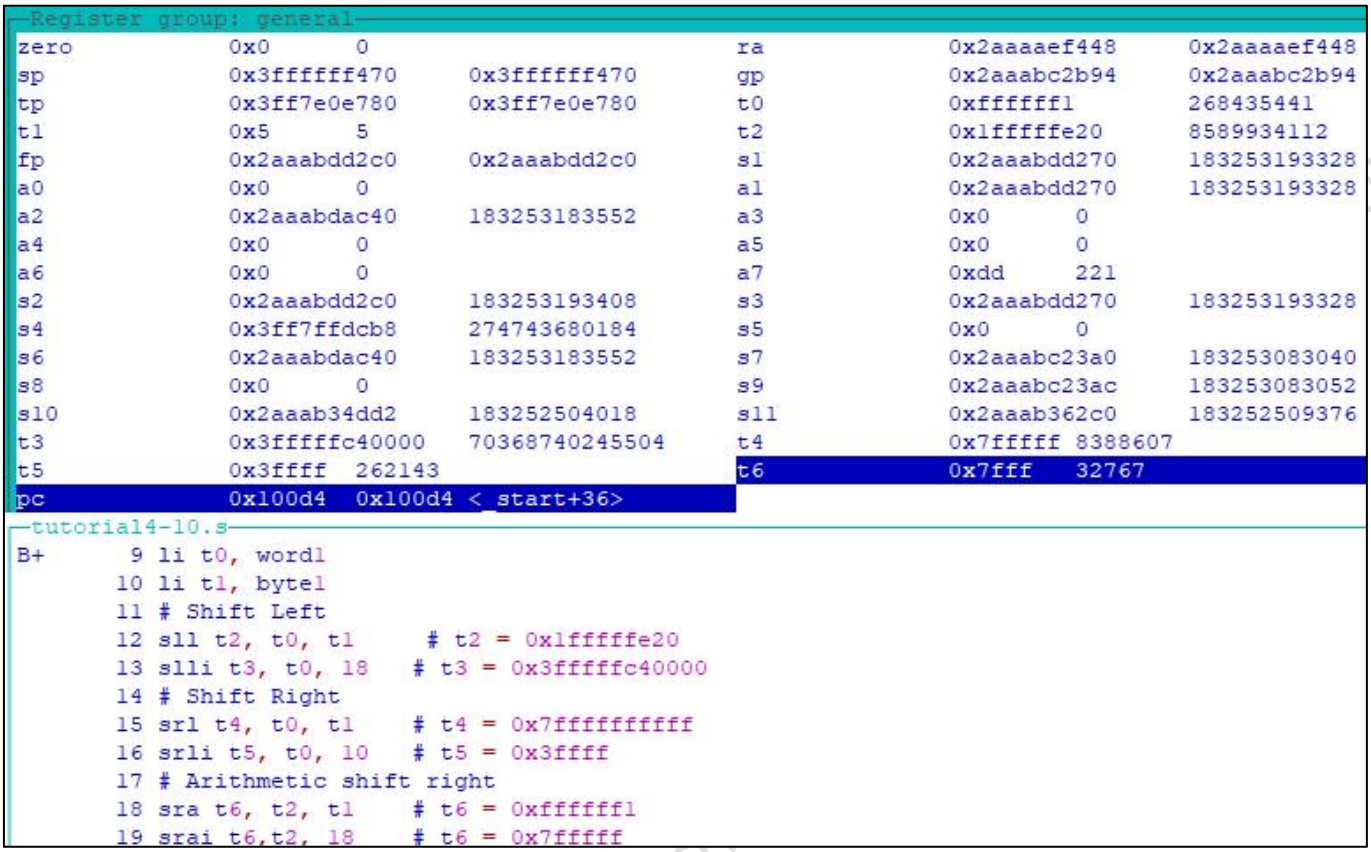
```
/*Listing 4-10 Shift operations*/

.section .data
```

[39] *Using a larger value such as* `srli t5, t0, 64` *gives an error message such as "Error: improper shift amount (64)" by the GNU assembler.*

```
# Define data
.equ word1,0x0ffffff1
.equ byte1,0x5
.section .text
.global _start
_start:
li t0, word1
li t1, byte1
# Shift Left
sll t2, t0, t1     # t2 = 0x1fffffe20
slli t3, t0, 18   # t3 = 0x3fffffc40000
# Shift Right
srl t4, t0, t1    # t4 = 0x7fffff
srli t5, t0, 10   # t5 = 0x3ffff
# Arithmetic shift right
sra t6, t2, t1    # t6 = 0xffffff1
srai t6,t2, 18    # t6 = 0x7fff
addi a7, x0, 93
ecall
```

Currently there is no rotate instruction. A GDB trace is shown below:

Figure 4-10 GDB trace of Listing 4- 10



4.5 Logical Instructions

RISC-V includes the following family of logical instructions:

- AND
- OR
- XOR
- NOT

These instructions are summarized in Table 4-7.

Table 4-7 RISC-V Logical Instructions

Instruction	Description	Syntax	Example
AND	Performs rs1 and rs2 bitwise `AND` operation, placing the result in rd	`and rd, rs1, rs2`	`and a0, t0, t1`
ANDI	Performs rs1 and sign-extended immediate field bitwise `ANDI` operation, placing the result in rd	`andi rd, rs1, imm`	`andi a1, a0, 0xf`
OR	Performs rs1 and rs2 bitwise `OR` operation, placing the result in rd	`or rd, rs1, rs2`	`or a2, t0, t1`
ORI	Performs rs1 and sign-extended immediate field bitwise `OR` operation, placing the result in rd	`ori rd, rs1, imm`	`ori a3, a2, 0x000`
XOR	Performs rs1 and rs2 bitwise `XOR` operation, placing the result in rd	`xor rd, rs1, rs2`	`xor a4, t0, t3`
XORI	Performs rs1 and sign-extended immediate field bitwise `XOR` operation, placing the result in rd	`xor rd, rs1, imm`	`xori a5, a2, 0xa`
NOT	Performs bitwise inversion of the bits in rs1 placing the result in rd	`not rd, rs1`	`not a5, a5`

Listing 4- 11 shows the result of the various logical instructions on RV64

Listing 4-11 Logical Instructions (RV64)

```
/*Listing 4-11 Logical operations (RV64 system)*/
.section .data
# Define data
.equ word1,0xaa55aa55
.equ maskupper,0xfff
.equ masklower,0x0
.equ xormask1, 0xaaaaaaaa
.equ xormask2, 0x55555555
.section .text
.global _start
_start:
        li t0, word1
        li t1, maskupper
```

```
        li t2, masklower
        li t3, xormask1 # t3 sign-extended = 0xffffffffaaaaaaaa
        li t4, xormask2
# and
and a0, t0, t1     # a0 =  0xa55, note Boolean algebra X AND 1 = x, X AND 0 = 0
        and a0, t0, t2     # a0 = 0, note X AND 0 = 0
        andi a1, a0, 0xf   # a1 = 0, since X = a0 = 0
# OR
        or a2, t0, t1     # a2 = 0xaa55afff, note Boolean or X or 1 = 1, X or 0 = X
        or a2, t0, t2     # a2 = 0xaa55aa55, X or 0 = X
        ori a3, a2, 0x000 # a3 = 0xaa55aa55
# XOR
        xor a4, t0, t3     # a4 = 0x00ff00ff, Note x XOR x = 0, x XOR xinverse = 1
        xor a4, t0, t4     # a4 = 0xff00ff00
        xori a5, a2, 0xa   # a5 = 0xaa55aa5f
# NOT
        not a5, a5 # a5 = 0xffffffff55aa55a0
        addi a7, x0, 93
        ecall
```

4.5.1 Logical function observations

- The AND function can be used to clear bits by *anding* the corresponding bit position with a binary zero.
    - Bits can be tested to see if they are high or low by anding with a binary one.
        - A non-zero value denotes that the corresponding bit tested was a binary one
        - A zero value denotes that the corresponding bit tested was a binary zero
- With the OR function, bits can be set by *oring* the corresponding bit position with a binary one
    - Bits can be tested to see if they are high or low by oring with a binary zero
        - A non-zero value denotes that the corresponding bit tested was a binary one
        - A zero value denotes that the corresponding bit tested was a binary zero
- Exclusive OR can check to see if the corresponding bit has equal polarity
    - A non-zero value indicates that the bit was of the opposite polarity
    - A zero value indicates that the bit was if the same polarity
    - Applying the exclusive or function using the same bit pattern as the number itself will clear the bits

4.6 Exercises for chapter 4

1. Write code to perform multiplication by 24 using shift instructions; do not use RISC-V multiply instruction variants. How would the memory layout look when storing two doublewords in memory on an RV64 system?

4.7 RISC-V instructions covered in chapter 4

Arithmetic Instructions (Base ISA)

- **ADD** – Add (32-bit)
- **ADDW** – Add word (sign-extended to 64-bit)
- **ADDI** – Add immediate
- **SUB** – Subtract (32-bit)
- **SUBW** – Subtract word

Multiply and Divide Instructions (RVM Extension)

- **MUL**– Multiply
- **MULH** – Multiply high (signed × signed)
- **MULHSU** – Multiply high (signed × unsigned)
- **MULHU**– Multiply high (unsigned × unsigned)
- **MULW** – Multiply word (32-bit)
- **DIV** – Divide (signed)
- **DIVU**– Divide unsigned
- **REM** – Remainder (signed)
- **REMU**– Remainder unsigned
- **DIVW** – Divide word
- **REMW** – Remainder word

Shift Instructions

- **SLI** – Shift left logical
- **SRL** – Shift right logical
- **SRA** – Shift right arithmetic
- **SLLW** – Shift left logical word
- **SRLW**– Shift right logical word
- **SRAW**– Shift right arithmetic word

Logical instructions

- **AND** – Bitwise AND
- **OR** – Bitwise OR
- **XOR** – Bitwise XOR
- **ANDI**– AND immediate
- **ORI** – OR immediate
- **XORI** – XOR immediate
- **NOT**– Bitwise NOT (pseudo-instruction)

5 Loops, Branches and Conditions

Overview of the chapter

Chapter 5 introduces control flow mechanisms in RISC-V assembly. It explains how to make decisions, repeat code (loops), and redirect program execution using conditional and unconditional branches.

5.1 J-Type and B-Type instructions

Chapter two discussed the control-transfer J and B-type instructions. Recall that unconditional jumps are J-type and conditional branches are B-type. The ability to vary program flow based on conditions such as greater than (>), less than (<) or equality greatly enhances the power of computing devices. RISC-V can perform conditional branches with a single instruction. Other instruction sets may use two instructions: first, performing a comparison, and then deciding whether to branch based on the status of a condition-code register flag.

Comparison using two instructions -

```
cmp r1, r2 # Compare two registers
bgt <label> # Branch if the value of register1 is greater than the value of register2
```

RISC-V only uses a single instruction -

```
bgt t0, t1, <label> # Branch if t0 is less than t1
```

This can result in more economical code.

5.1.1 B-Type instruction details

Consider the instruction `blt t0, t1, exit` where the branch instruction is located at 0x100c0 and the label <exit> is located at location 0x100c8. When t0 is less than t1 then the flow will branch to the address at <exit>. The opcode in this case is 0x0062c463.

Referring to Figure 5-1 the diagram shows that the offset has a value of 8 which is the number of places that the program will branch to (0x100c8 minus 0x100c0). Recall that bit zero need not be encoded in the immediate value which specifies the offset and is always implicitly set to zero. This means that the offset is always even. The reason that the offset is a multiple of two rather than four is to accommodate RISC-V 16-bit implementations. The register operands use the X register numbers, showing the values 5 and 6 which correspond to registers t1 and t0.

Figure 5- 1Breakdown of blt instruction

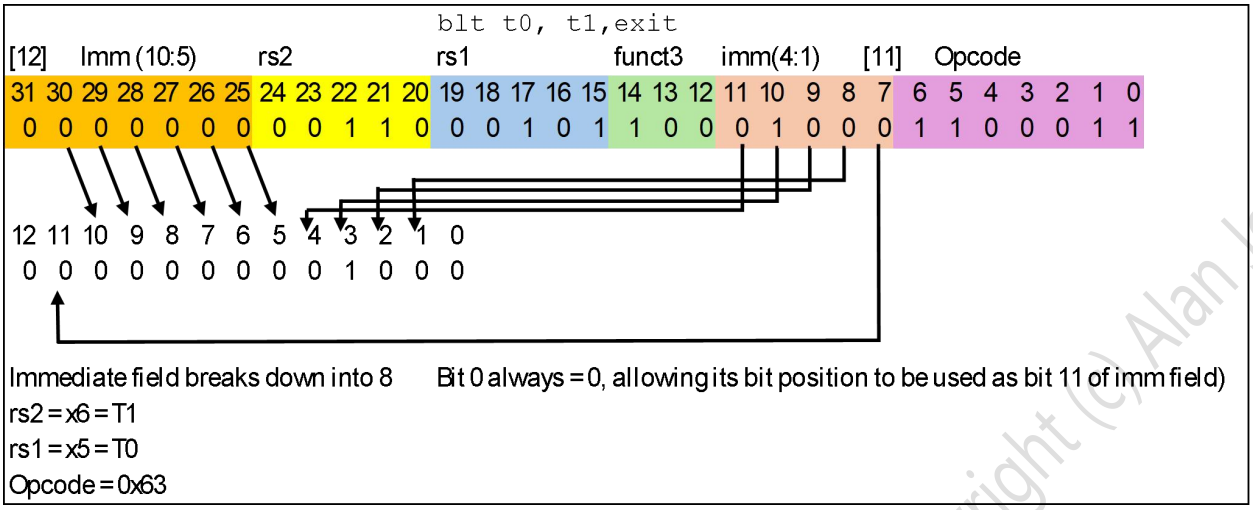


Table 5- 1 shows the available branch instructions, including the additional pseudo-instructions.

Note that comparisons can be made with signed and unsigned values.

Table 5-1 Conditional branch instructions

Instruction	Description	Syntax	Example
BLT	Branch if less than	`blt rs1, rs2, imm`	`blt t0, t1, exit`[40]
BLTU	Branch if less than unsigned	`bltu rs1, rs2, imm`	`bltu t0, t1, exit`
BLTZ	Branch if less than zero*	`bltz rs1, imm`	`bltz t0, exit`
BLE	Branch if less than or equal to zero*	`ble rs1, rs2, imm`	`ble t0, t1, exit`
BLEU	Branch if less than or equal unsigned*	`bleu rs1, rs2, imm`	`bleu t0, t1, exit`
BLEZ	Branch if less than or equal to zero*	`blez rs1, imm`	`blez t0, exit`
BGE	Branch if greater than or equal	`bge rs1, rs2, imm`	`bge t0, t1, exit`
BGT	Branch if greater than*	`bgt rs1, rs2, imm`	`bgt t0, t1, exit`
BGTU	Branch if greater than unsigned*	`bgtu rs1, rs2`	`bgt t0, t1, exit`
BGTZ	Branch if greater than zero*	`bgtz, rs1, imm`	`bgtz t0, exit`

[40] *This is a memory location pointed to by the label "exit"*

Instruction	Description	Syntax	Example
BGEU	Branch if greater than or equal to zero unsigned	`bgeu rs1, rs2, imm`	`bgeu t0, t1, exit`
BGEZ	Branch if greater than or equal to zero*	`bgez rs1, imm`	`bgez t0, exit`
BEQ	Branch if equal	`beq rs1, rs2, imm`	`beq t0, t1, exit`
BEQZ	Branch if equal to zero*	`beqz rs1, imm`	`beqz t0, exit`
BNE	Branch if not equal	`bne rs1, rs2, imm`	`bne t0, t1, exit`
BNEZ	Branch if not equal to zero*	`bnez rs1, imm`	`bnez t0, exit`

*=Pseudo instruction

5.1.2 J-Type instruction details

5.1.2.1 JAL

The format of the Jump and Link instruction (JAL) is `JAL rd, <label>`.

The instruction `jal makesquare` is equivalent to the pseudo instruction `j makesquare`, which disassembles to the non-aliased instruction `jal ra,<makesquare>`. It has a 20-bit immediate value specifying bits 20:1. Bit 0 of the immediate value is not coded and is always set to zero to give even values. This gives a total of 21 *signed* bits, which is equivalent to a range of minus one MB through to plus one MB.

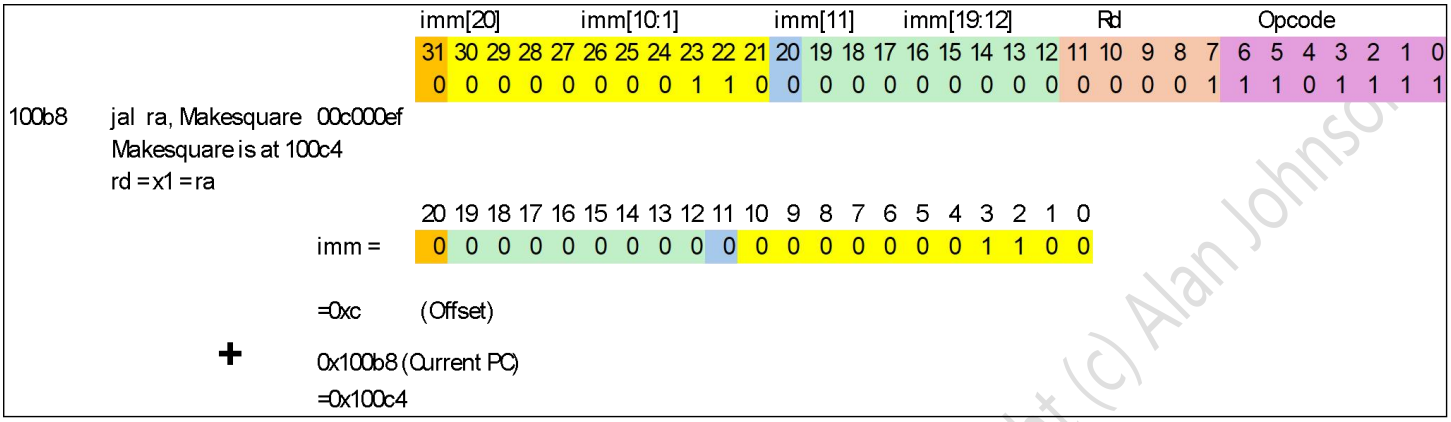
By convention, the destination register (rd) is register X1 (ra), if no destination register is specified; ra is automatically used. An instruction such as `jal ra, makesquare` will use an offset to the address located at the label <makesquare>. The return address register (ra) will hold the address of the next instruction following the current `jal ra, makesquare` instruction (current PC+4 = 0x100bc).

An instruction such as `jal zero, mylabel` will not store the return address[41] and is an unconditional jump. The aliased instruction `J` actually expands to `jal x0, offset`.

Referring to Figure 5-2 the immediate value is 0xc so adding this value to the address of the JAL instruction (here 0x100b8) gives a jump address of 0x100c4 which is where the makesquare routine is located. When the routine has finished, the program flow returns to the address stored in the ra register (0x100bc). This is achieved by the pseudo instruction `ret` which is an alias for `jalr, zero,0(ra).`

[41] *Since the zero register is not writable.*

Figure 5-2 Bit breakdown of JAL instruction



5.1.2.2 JALR

The jump and link register instruction (JALR) gets its target address by adding a sign-extended 12 -bit value to the source register rs1, setting the least significant bit to zero. The destination register will be loaded with the address of the instruction following the JALR instruction.

`JALR zero, 0(ra),` will return to the address in the ra register. The ra register is not updated in this case since the X0 register has been specified as rd. The pseudo instruction for `jalr rd,offset(rs1)` is `jr`.

The `ret` instruction is the pseudo instruction for `jalr x0,0(x1).`

5.1.2.3 Difference between jr and ret

The pseudo-instruction `ret` will map `to jalr x0,0(ra)` but `jalr` is free to use different registers since it has the form `jalr rd,offset(rs1)` so an instruction such as `jalr, offset(t0)` is acceptable.

5.1.3 Implementing a loop counter to square numbers

The first example is that of a simple loop counter. The program uses a subroutine to compute squares of numbers from 1 to 20. The results are stored in consecutive halfword locations. The listing features one unconditional branch (`blt`) and one unconditional jump (`jal`). After the subroutine has completed, the `jalr` instruction will jump to the instruction (`addi a7,zero,93`) immediately following the instruction (`jal squareit`) that called the subroutine. When tracing the program flow with GDB, use S(tep) rather than N(ext), since "N" will skip a function such as `squareit`.

Listing 5-1 Squaring numbers from 1 to 20

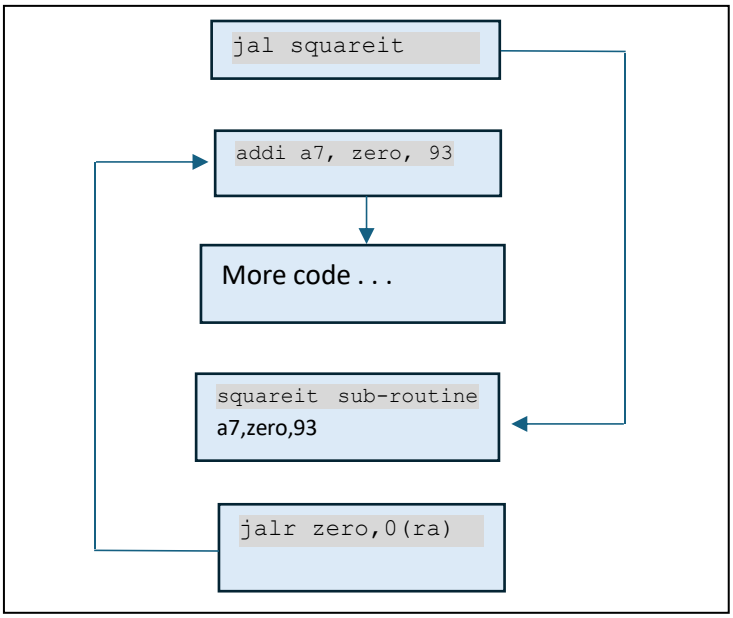
```
# listing5-1

.section .text

.global _start

_start:

addi   t0,zero,21  # Set up counter
```

```
addi   t1,zero,1 # Start at 1
la a0, storesquares
jal squareit # Jump to routine at <squareit>, saving return address in ra
addi   a7,zero,93 # Routine finished, time to leave
ecall
squareit:
mul    t2,t1,t1 # Square the contents of t1 and put the result in t2
sh t2,0(a0)
addi a0, a0,2 # Point to the next halfword location (2 bytes on)
addi   t1,t1,1 # increment the number to be squared
blt    t1,t0,squareit # If 20 numbers have been squared then return from routine
jalr   zero,0(ra)
.section .data
storesquares:
.space 64
```

Examining the memory after the subroutine has finished shows -

```
0x1111c:      0x0001 0x0004 0x0009 0x0010 0x0019 0x0024 0x0031 0x0040
0x1112c:      0x0051 0x0064 0x0079 0x0090 0x00a9 0x00c4 0x00e1 0x0100
0x1113c:      0x0121 0x0144 0x0169 0x0190
0x1111c:      1      4      9      16     25     36     49     64
0x1112c:      81     100    121    144    169    196    225    256
0x1113c:      289    324    361    400
```

Figure 5- 3 Program flow of makesquares listing



1. Call subroutine <squareit>
   Return from subroutine
   Resume code execution

J-Type (Unconditional Jumps)

JAL Jump and link

JALR Jump and link register

5.1.4 Summary of jump instructions

jr is a *jump* instruction

and is a pseudo instruction for:

```
jalr x0, rs1, 0
```

which is:

- Jump to the address in rs1
- Do **not** write a return address (because rd = x0)

`ret` is a *pseudo instruction* for returning from a function, expanding to:

ret → jalr x0, ra, 0

which is:

- Jump to the address stored in ra (x1)
- Do **not** write a return address

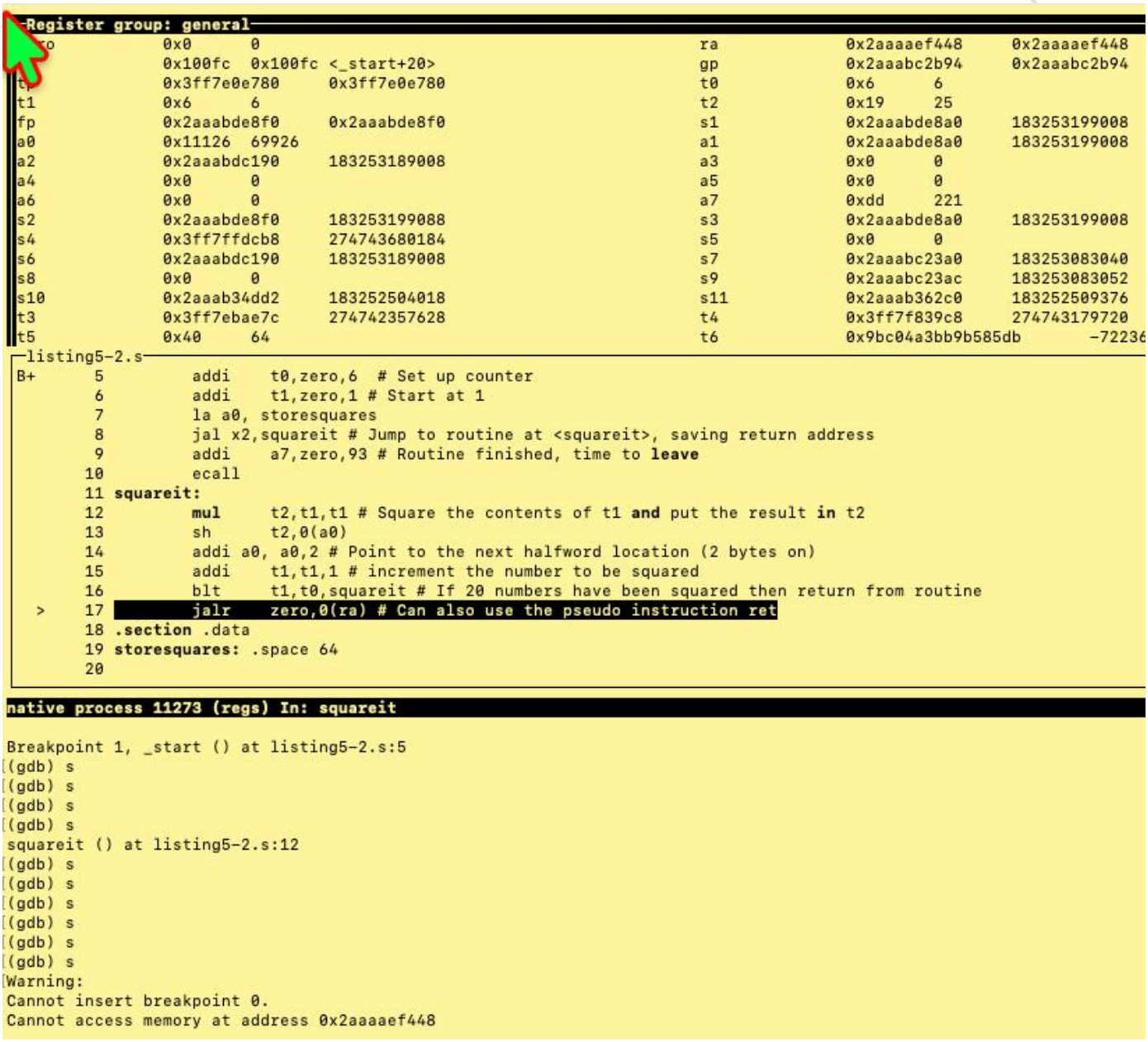
So ret **is a special case of** `jr` where the register is fixed to x1 (ra).

`jr rs1` `jalr x0, rs1, 0` Jump to an arbitrary register

`ret` `jalr x0, ra, 0` Return from a function

5.2 Exercises for chapter 5

1. Write a program that takes as its input a number less than 1000 and then calculates the number of primes below that number.
2. The program crashes after executing the `jalr zero,0(ra)` instruction highlighted in the GDB trace below – why?



```
Register group: general
zero           0x0         0                                  ra             0x2aaaaef448     0x2aaaaef448
               0x100fc     0x100fc <_start+20>                gp             0x2aaabc2b94     0x2aaabc2b94
t              0x3ff7e0e780     0x3ff7e0e780                  t0             0x6         6
t1             0x6         6                                  t2             0x19        25
fp             0x2aaabde8f0     0x2aaabde8f0                  s1             0x2aaabde8a0     183253199008
a0             0x11126     69926                              a1             0x2aaabde8a0     183253199008
a2             0x2aaabdc190     183253189008                  a3             0x0         0
a4             0x0         0                                  a5             0x0         0
a6             0x0         0                                  a7             0xdd        221
s2             0x2aaabde8f0     183253199088                  s3             0x2aaabde8a0     183253199008
s4             0x3ff7ffdcb8     274743680184                  s5             0x0         0
s6             0x2aaabdc190     183253189008                  s7             0x2aaabc23a0     183253083040
s8             0x0         0                                  s9             0x2aaabc23ac     183253083052
s10            0x2aaab34dd2     183252504018                  s11            0x2aaab362c0     183252509376
t3             0x3ff7ebae7c     274742357628                  t4             0x3ff7f839c8     274743179720
t5             0x40        64                                 t6             0x9bc04a3bb9b585db       -72236

listing5-2.s
B+      5         addi    t0,zero,6  # Set up counter
        6         addi    t1,zero,1 # Start at 1
        7         la a0, storesquares
        8         jal x2,squareit # Jump to routine at <squareit>, saving return address
        9         addi    a7,zero,93 # Routine finished, time to leave
       10         ecall
       11 squareit:
       12         mul     t2,t1,t1 # Square the contents of t1 and put the result in t2
       13         sh      t2,0(a0)
       14         addi a0, a0,2 # Point to the next halfword location (2 bytes on)
       15         addi    t1,t1,1 # increment the number to be squared
       16         blt     t1,t0,squareit # If 20 numbers have been squared then return from routine
  >    17         jalr    zero,0(ra) # Can also use the pseudo instruction ret
       18 .section .data
       19 storesquares: .space 64
       20

native process 11273 (regs) In: squareit

Breakpoint 1, _start () at listing5-2.s:5
(gdb) s
(gdb) s
(gdb) s
(gdb) s
squareit () at listing5-2.s:12
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
Warning:
Cannot insert breakpoint 0.
Cannot access memory at address 0x2aaaaef448
```

5.2.1 RISC-V jump and branch instructions covered in chapter 5

B-Type (Conditional Branches)

BLT – Branch if less than

BLTU – Branch if less than unsigned

BLTZ– Branch if less than zero *(pseudo-instruction)*

BLE – Branch if less than or equal *(pseudo-instruction)*

BLEU – Branch if less than or equal unsigned *(pseudo-instruction)*

BLEZ – Branch if less than or equal to zero *(pseudo-instruction)*

BGE – Branch if greater than or equal

BGT – Branch if greater than *(pseudo-instruction)*

J-Type (Unconditional Jumps)

JAL– Jump and link

JALR – Jump and link register

6 The Stack, Macros and Functions

Overview of the chapter

Chapter 6 focuses on **modularizing code** in RISC-V assembly using **functions, macros, and the stack**. It introduces structured programming principles in low-level development and shows how to organize code effectively.

6.1 Overview

The concepts between macros and functions are similar, but the way the programs are assembled leads to tradeoffs in performance and code size. The previous chapter used a subroutine called `<squareit>`. The routine can be a separate piece of code outside of the main listing, which means that routines can be used as functions that other programs can call,rather than having to keep writing the additional code, enhancing clarity and manageability.

6.1.1 The Stack

Functions will make use of the stack. In general, the stack is a data structure that stores data in a structured manner. As an example, a register's contents can be *Pushed* on to the stack and can be restored by popping the data from the stack back to the register again. Push and Pop operations are performed in a *Last in First out (LIFO)* manner, in that if multiple items were pushed on to the stack the last item pushed would be the first one restored. The stack is a location in memory. The *stack pointer* shows where in memory the lowest address of the stack is situated. When data is pushed, the stack pointer will be decremented to a lower memory location, and when data is popped, the stack pointer will be incremented.

There are some subtle differences in the RISC-V stack implementation -

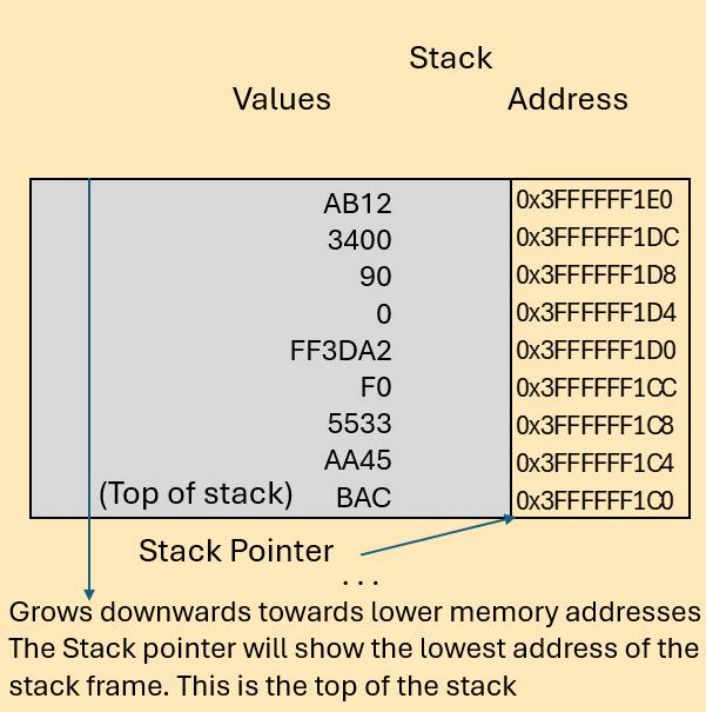
Note that RISC-V does not use actual push and pop instructions that are found in other processor architectures. A push to the stack is accomplished using the store instruction, and a pop is accomplished using the load instruction. This means that the stack can be *randomly* accessed.

Both these load and store instructions are familiar; the only difference is that the stack pointer is used as the operand rather than a normal register. With RISC-V, the convention is to use register X2 as the stack pointer; its ABI name is sp. RV64[42] architectures require that the stack be 16-byte (128-bit) *aligned*. The stack, by default with RISC_V grows downwards and is termed *a full descending stack.*

At program entry, the stack should already be 16-byte aligned on 64-bit systems.

[42] *Also RV 32*

Figure 6- 1 Stack contents operations



The example program shows how to allocate stack space, followed by pushing (storing) and popping (loading) items using the stack.

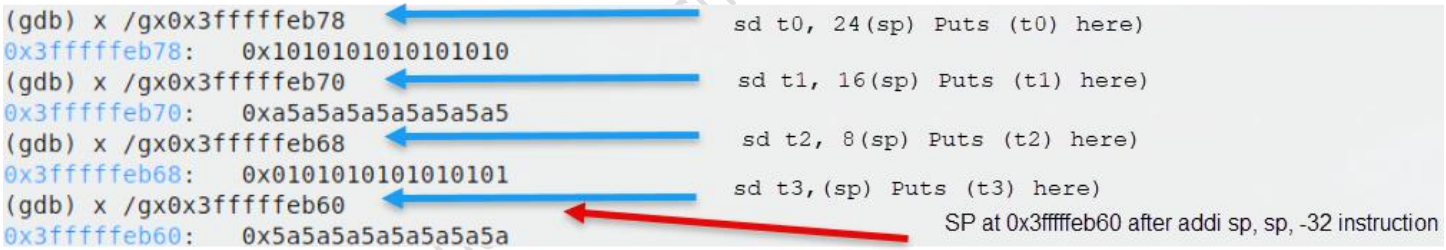
```
Listing  6-1 Allocation and deallocation of the stack

# Listing 6-1

.section .text

.global _start

_start:

# Allocate 32 Bytes for the stack

    addi sp, sp, -32

    li t0,0x1010101010101010

    li t1,0xa5a5a5a5a5a5a5a5

    li t2,0x0101010101010101

    li t3, 0x5a5a5a5a5a5a5a5a

#Push registers

    sd t0, 24(sp)

    sd t1, 16(sp)

    sd t2, 8(sp)
```

```
    sd t3,(sp)
#Pop registers, in a LIFO fashion
    ld t3, (sp)
    ld t2, 8(sp)
    ld t1, 16(sp)
    ld t0, 24(sp)
# Clean up stack
    addi sp,sp,32
exit:
    li a7, 93
    ecall
```

Comments.

- Stack pointer initially = 0x3FFFFFEB80
- After `addi sp,sp,-32`= 0x3FFFFFEB60

GDB can be used to view the stack contents–



Note use /g with gdb.

Each location shown is a doubleword.

Memory Address	Contents
0x3FFFFFEB78	0x1010101010101010
0x3FFFFFEB70	0xa5a5a5a5a5a5a5a5
0x3FFFFFEB68	0x0101010101010101
0x3FFFFFEB60	0x5a5a5a5a5a5a5a5a

6.1.2 Functions

Functions are used to promote coding efficiency and clarity. They are sections of code that can be included in a program and shared with others as *libraries*. Over time, a coder will usually generate their own functions for use in their code. When using external functions, registers can be saved on the stack prior to calling the function, thus ensuring that on return from the function code everything has been restored and coding will continue from where it left off. The Program Counter (PC) keeps track of the location in memory where the code is next to be executed. When a portion of code calls a function, it is termed the *caller*. The code that was called (the function itself) is termed the *callee*. When calling a function, there are several tasks that the caller must perform, and similarly the callee has its own responsibilities. When a function calls another function, the ra register must be preserved; otherwise, the original return address used by the first calling routine will be lost.

The registers follow certain conventions which are described below:

1. There are eight argument registers, a0-a7
2. Additional arguments are pushed onto the stack and popped by the called routine.
3. Two registers are used for return values – a0 and a1
4. More values will use a reference to an address (call by reference) where the additional data is stored.
5. Values equal to double the XLEN bits can be passed using two registers. The low-order XLEN bits are passed in the lowest-number register, such as a0 and the high-order XLEN bits are passed in the higher register, such as a1
6. The value can be passed on the stack.
7. If there is only one register available, then it can be used in conjunction with the stack.
8. Nested functions *must* preserve the ra register,
9. *Leaf* functions [43] do not need to save the return address to the stack,
10. When functions are called without knowing the register usage then the rules shown in Table 2-2 must be respected[44].

6.2 Calling nested routines

This program uses two routines. The first routine calls the second routine, which simply returns flow back to its caller, which then, in turn, returns flow back to the main program. The program outputs text to illustrate the location. Note that the parent routine must save the ra register prior to calling the child routine, as the jump will cause its value to be overwritten. This is not the case for the child routine, as it is a leaf function.

[43] *A leaf function is a function that has been called but does not call any other functions.*

[44] *A summary of the rules –*

Zero register (x0) is immutable,

ra must be preserved,

t0-t7 If needed should be saved by the calling routine,

s0-s11 Saved by the callee if used via the stack,

a0-a7 If needed should be saved by the calling routine,

The main routine saves the argument registers for printing a second time when it has returned from the parent and child routines.

Listing 6-2 Nested routines example.

```
.data

    # This program shows nested functions where the main routine calls a routine, which in turn
calls another routine

mainmessage:    .ascii "In main program\n"

parentmessage: .ascii "Now in Parent Routine\n"

childmessage:  .ascii "Now in child routine\n"

.equ mainlen, 16

.equ parentlen, 22

.equ childlen, 21

.text

.global _start


_start:

       .option norelax

       li a0, 1

    la a1, mainmessage

    li a2, mainlen

    #Set up stack

    addi sp,sp,-32 #Allocate

    sw a0, 0(sp)

    sw a1, 8(sp)

    sw a2, 16(sp)

    li a7,64

    ecall

    jal ra, parent

    lw a0, 0(sp)

    lw a1, 8(sp)

    lw a2,16(sp)

    addi sp,sp,32

    li a7,64
```

```
    ecall
exit:
    li a7,93
    ecall

parent:
# parent is not a leaf function as it calls the routine "child"
    li a0,1
    la a1, parentmessage
    li a2, parentlen
    li a7,64
    ecall
    sw ra, 24(sp)
    jal ra, child
    lw ra, 24(sp)
    ret

child:
# child is a leaf function as it does not call any other routines
    li  a0, 1           # Set start value of 1
    la a1, childmessage
    li a2, childlen
    li a7, 64
    ecall
    ret
```

6.2.1 Combining separate programs

The next program (`maina.s`) calls an external program (`squareit.s)` to calculate the squares.

Listing 6-3 main.s

```
.section .data
message:   .ascii "\nPlease enter a sequence of digits (up to 4   characters)to       be squared\n"
  .equ messagelength, 70
  errormessage1:  .ascii "\nIllegal character(s) found, please enter   only base10 numbers\n"
```

```
.equ errormessage1length, 63
  errormessage2:  .ascii "\nIncorrect number of digits, please enter 1,2,3 or 4 digits\n"
  .equ errormessage2length, 60
  inputbuffer:    .space 16 # Holds user input
  numberbuffer:   .space 16 # Holds the converted ASCII to integer numbers
  asciioutputbuffer: .space 16
  successmessage: .ascii "\nThe result is "
  .equ successmessagelength, 16
  tidyupchars: .ascii "\n\n"
  .equ tidyupcharslength, 2
  .equ linefeed, 10
.section .text
  .global _start
_start:
# Prompt for number
  li a0, 1 #<stdout>
  la a1, message
  li a2, messagelength
  li a7, 64
  ecall
# Read in number from keyboard
  li a0, 0 # file descriptor 0 (stdin)
  la a1, inputbuffer # address of the buffer
  li a2, 5 # Max number of bytes to read
  li a7, 63 # Read syscall
  ecall
# Convert inputted ASCII number to decimal
# Load the address of the ASCII number string
  la a0, inputbuffer      # a0 = address of "string"
  li a1, 0                # a1 = result (initialize it to 0)
  li t0, linefeed # Linefeed character
 li a3, 48       # Ascii number is actual number +48 so need to subtract this value
```

```
  li a4, 10       # used in convert_loop to multiply input character to correct position
  li a5, 57       # upper bound (entered digit can't be > 9)
  li a6, 48       # lower bound (entered digit can't be < 0)
  la t5, inputbuffer+5    # Check for too many digits entered
convert_loop:
# Load the next ASCII character
  lb a2, 0(a0)            # a2 = *a0 (current ASCII character)
  beq t0, a2,skip_valuechecks #(if <LF> then skip the checks for legal decimal number)
  bgt a2,a5,illegalcharacter  #(too high)
  ble a2, a6,illegalcharacter #(too low
skip_valuechecks:
# Check if we've reached the <LF> character (end of input string)
  beq t0, a2, conversion_done # If character is <LF> then all numbers have been processed
# Convert ASCII to integer:
  li a3, 48         # Load '0' ASCII value
  sub a2, a2, a3    # Convert ASCII character to integer
  mul a1, a1, a4   # shift left by one decimal place
  add a1, a1, a2  # Add the digit to result
# Move to the next character in the string
  addi a0, a0, 1   # Increment the pointer
  beq t5,a0, toomanydigits # More than 4 digits have been entered
  j convert_loop # Next character
conversion_done:
  mv a0,a1
  jal squarenumber
# The number has been squared, time to convert back to ASCII format
  la a1, asciioutputbuffer +11   # Point to the end of buffer
  sb zero, 0(a1)        # Null-terminate the string
  li t3, 0
convertbacktoascii:
  addi t3, t3, 1
  li t1, 10             # Load divisor (10)
```

```
  rem t2, a0, t1        # Get last digit (a0 % 10)
  div a0, a0, t1        # Remove last digit (a0 / 10)
  addi t2, t2, 48       # Convert digit to ASCII (for printing)
  addi a1, a1, -1       # Move buffer pointer back one place
  sb t2, 0(a1)          # Store ASCII character in buffer
  bnez a0, convertbacktoascii # Repeat if number is not zero
printsuccess:
  li a0, 1              # syscall for print_string
  la a1, successmessage         # Address of ASCII string
  li a2, successmessagelength
  li a7, 64              # Syscall number for printing string
  ecall                 # Make syscall
print:
  li a0, 1              # syscall for print_string
  la a1, asciioutputbuffer             # Address of ASCII string
  li a2,12
  li a7, 64              # Syscall number for printing string
  ecall                 # Make syscall
  li a0, 1
  la a1, tidyupchars
  li a2, tidyupcharslength
  li a7, 64
  ecall
exit:
  li a7, 93             # Syscall for exit
  ecall
illegalcharacter:
  li a0, 1
  la a1, errormessage1
  li a2, errormessage1length
  li a7, 64
  ecall
```

```
  j exit
toomanydigits:
  li a0, 1
  la a1, errormessage2
  li a2, errormessage2length
  li a7, 64
ecall
  j exit
```

Listing 6-4 squareit.s

```
.global squarenumber
squarenumber:
# Note Register a0 contains the user input (the number to be squared)
# The same register will hold the return value
mul   a0,a0,a0 # Square the contents of a0 and put the result in a0
jalr  zero,0(ra)
```

The `makefile` is as shown

Makefile for squareit

```
OBJECTS = maina.o squareit.o
all: maina
%.o: %.s
	as -mno-relax $< -g -o $@
maina: $(OBJECTS)
	ld -o maina $(OBJECTS)
```

Copy the listings to the programs named in the makefile:

```
$ cp listing6-3.s maina.s
$ cp listing6-4.s squareit.s
$ make
as -mno-relax maina.s -g -o maina.o
as -mno-relax squareit.s -g -o squareit.o
ld -o maina maina.o squareit.o
```

Validate the program through gdb or running it directly from the command line.

```
./maina
Please enter a sequence of digits (up to 4 characters) to be squared
6
The result is
36
./maina
Please enter a sequence of digits (up to 4 characters) to be squared
12
The result is
144
./maina
Please enter a sequence of digits (up to 4 characters) to be squared
843
The result is
710649
./maina
Please enter a sequence of digits (up to 4 characters) to be squared
9999
The result is
99980001
$ ./maina
Please enter a sequence of digits (up to 4 characters) to be squared
3f
Illegal character(s) found, please enter only base10 numbers
./maina
Please enter a sequence of digits (up to 4 characters)to be squared
23146
Incorrect number of digits, please enter 1,2,3 or 4 digits
```

Note that the *called* program `<squareit.s>` has declared <squarenumber> as a *global*, this is to allow it to be shared and used by other files. Declarations without the .global directive are treated as *local* to the file in which they were declared and are not accessible by other

programs.

When tracing the program flow with GDB, set the first breakpoint to `b _start` rather than `b 1`.

The programs are named `maina.s` and `squareit.s.`

The program flow is:

1. Prompt the user to enter a number
2. Get the number from the keyboard entry, storing it in *inputbuffer*
3. Convert the number from its ASCII representation to integers, storing it in *numberbuffer*
4. Validate the number to be in the range <0-9> unless the ASCII character is Linefeed[45], If not valid, then print out an error message (errormessage1) and exit.
5. Validate the quantity of digits entered, allowing only 1,2,3 , if invalid, print out an error message (errormessage2) and exit.
6. The convert_loop routine will place the converted integers in the correct buffer location; the multiplication by ten shifts the digit to the correct magnitude value.[46]
7. Once the <linefeed> character has been encountered, then all digits have been converted.
8. The number is passed to the squarenumber routine in the program squareit. The squared number will be returned via register a0[47].
9. The next step is to display the result by converting the squared integer back to ASCII format which is performed by the routine convertbacktoascii.
10. Finally, the result is printed on the screen, and the program exits.

The program should be stepped through with GDB; ensure that the routines - `convert_loop`: and `conversion_done`: are fully understood.

Further example –

Listing 6-5 Callerprogram

```
# This is the caller program (callerprogram.s) that passes the address of a string
# to be printed to another program (calledprogram.s)
.section .data
message:
    .asciz "This string was defined in the calling program!\n"
    .section .text
```

[45] *Pressing enter on the keyboard will store the linefeed character (0xa) in the buffer*

[46] *For example, the number 6543 will be processed in stages as 6, 60, 65, 650, 654, 6540, 6543 by the multiply and add instructions in convert_loop*

[47] *It has not been necessary to store values on the stack in this particular example. If the called routine were to overwrite any register that would need to be preserved then the caller/callee conventions would be respected.*

```
    .global _start
    .extern print_str # Not defined here, defined externally
_start:
    la   a0, message        # a0 = address of string
    call print_str   # call external routine
    # exit(0)
    li   a0, 0 # Return success, check in linux with echo $?
    li   a7, 93         # exit syscall
    ecall
```

Listing 6-6 Called program

```
# Callee program (calledprogram.s)
    .section .text
    .global print_str
print_str:
# Allocate and push stack items
    addi sp, sp, -16
    sd   ra, 8(sp)
    sd  s0, 0(sp) # Callee has the burden to save the S registers
    mv   s0, a0         # save pointer to the string which was loaded into a0
    # find string length
    mv   t0, s0
1:
    lbu  t1, 0(t0) # Put current character of string into t1
    beqz t1, 2f  # If t1 equals zero then we have reached the end of the string
    addi t0, t0, 1 # if not get the next character string
    j    1b # Jump backwards to label 1:
2: # OK we now have reached the end of the string with .asciz
    sub  a2, t0, s0     # length is now computed from t0
    mv   a1, s0         # buffer address loaded into a1
    li   a0, 1          # stdout
    li   a7, 64         # write syscall
    ecall
```

```
# Unwind the stack
    ld   ra, 8(sp)
    ld   s0, 0(sp)
    addi sp, sp, 16
    ret # Our work is done here!
```

Rename the programs.

```
$ cp listing6-5.s callerprogram.s
$ cp listing6-6.s calledprogram.s
```

Build with:

```
$ as -g -o callerprogram.o callerprogram.s
$ as -g -o calledprogram.o calledprogram.s
$ ld -o printstring callerprogram.o calledprogram.o
```

6.3 Macros

Macros, like functions can be used to promote coding efficiency and clarity. Macros can be included inline within a program or defined separately using the `.include` directive. Macro code is encased between the directives `.macro` and `.endm`. They are used to repeat frequently used instructions using different parameter values. The format of a macro is macroname argument1, argument2, . . . Inside the macro code, these arguments have a backslash \ character in front of them. Macros differ importantly from functions in that the actual macro code is substituted inline within the main code, this means that 100 calls to the same macro will generate 100 copies of the macro code. The use of Macro's can increase performance since there is no need to deal with return addresses as is the case for functions.

Check!

```
./printstring
This string was defined in the calling program!
```

Listing 6-7 Macro example (callmacro.s)

```
# This code calls a macro to print strings to stdout
# The input parameters are the string's location and its length
.section .data
string1:   .ascii "\nThis string was printed using a macro call"
string2:   .ascii "\nAnd so was this\n"
.equ stringlength1, 43
.equ stringlength2, 17
.section .text
```

```
.include "printmacro.s"
	.global _start
_start:
li a0, 1 #stdout
# Save a0 on to the stack, would have been simpler to just load it again after the macro call
# however this illustrates an example
# Allocate space on the stack
	addi sp, sp, -16
	sw a0, 12(sp)
	print string1, stringlength1
	lw a0, 12(sp)
# No need to preserve a0 this time since we no longer need to restore it
	print string2, stringlength2
# Exit program
	li a7, 93		# Syscall number for exit
	ecall			# Make syscall
```

Listing 6-8 called macro program (printmacro.s)

```
.macro print location, length
la a1, \location
li a2, \length
li a7, 64
ecall
.endm
```

Build the code and run it!

```
$ as -g -o listing6-7.o listing6-7.s
$ as -g -o printmacro.o listing6-8.s
$ ld -o printit printmacro.o listing6-7.o
$ ./printit

This string was printed using a macro call
```

The disassembly (below) shows that the macro has been placed in line, GDB shows that the address of string1 is located at 0x11128 and that string2's address is at 0x11153.

```
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x0000000000011128  __DATA_BEGIN__
0x0000000000011128  string1
0x0000000000011153  string2
0x0000000000011164  __SDATA_BEGIN__
0x0000000000011164  __bss_start
0x0000000000011164  _edata
0x0000000000011168  __BSS_END__
0x0000000000011168  _end
(gdb)
```

The next part of the macro is the `li` instruction, which loads the string length into register a2, finally, the syscall is invoked.

Note unlike functions, there are no return calls since the macro code is inline.

```
00000000000100e8 <_start>:
100e8:      00100513    li      a0,1
100ec:      ff010113    addi    sp,sp,-16
100f0:      00a12623    sw      a0,12(sp)
100f4:      00001597    auipc   a1,0x1
100f8:       03458593   addi    a1,a1,52 # 11128 <__DATA_BEGIN__>
100fc:      02b00613    li      a2,43
10100:      04000893    li      a7,64
10104:      00000073    ecall
10108:      00c12503    lw      a0,12(sp)
1010c:      00001597    auipc   a1,0x1
10110:       04758593   addi    a1,a1,71 # 11153 <string2>
10114:       01100613   li      a2,17
10118:       04000893   li      a7,64
10120:        05d00893    li      a7,93
10124:        00000073         ecall
```

The next macro is part of the same program and does not call the macro externally

Listing 6-9 nternal Macro used to print newline character for the squares program

```
.section .data
```

```
    .equ begincount,1
    .equ endcount,21
.section .rodata
newline:
    .byte 10
.section .text
.global _start
# Must define macros before they are called
.macro PRINT_NL nl
    # Print Newline Macro
    # Save all argument registers that are being used by the macro
    addi sp,sp,-32
    sd a0, 24(sp)
    sd a1,16(sp)
    sd a2, 8(sp)
    sd a7,0(sp)
    li a0, 1    # stdout
    la a1, newline   #Newline
    li a2, 1
    li a7, 64           # sys_write
    ecall
    ld a7,(sp)
    ld a2,8(sp)
    ld a1,16(sp)
    ld a0,24(sp)
    addi sp,sp,32
.endm
_start:
    li s0, begincount         # for i = 1 to 20
li s1, endcount           # if we reach 21 we need to leave!
forloop:
# Call the macro  (PRINT_NL) to print the newline character
```

```
PRINT_NL newline
beq s0, s1, exit    # if i == 21, exit
    mul s2, s0, s0      # Square the number in the counter s2 = i * i
    # Convert and Print Square
    mv a0, s2
    jal ra, print_integer
    addi s0, s0, 1      # count++
    j forloop
exit:
    li a0, 0            # Return 0 status
    li a7, 93           # sys_exit
    ecall
print_integer:
    addi sp, sp, - 32   # Allocate 32 bytes on stack
    addi t0, sp, 31
    li t1, 10           # Divisor in decimal system
    # Count the number of characters before returning zero when dividing, convert to ASCII and set
up syscall write parameters in A registers
conv_to_ascii:
    rem t2, a0, t1      # Get digit
    addi t2, t2, 48     # Convert to ASCII
    addi t0, t0, -1     # Move pointer back BEFORE storing
    sb t2, 0(t0)        # Place ascii char in the bottom of the stack
    div a0, a0, t1
    bnez a0, conv_to_ascii
    addi t1, sp, 31
    sub a2, t1, t0      # length in a2
    mv a1, t0           # buffer address in a1
    li a0, 1            # stdout
    li a7, 64           # sys_write
    ecall
    addi sp, sp, 32     # Free up stack
    ret
```

```
Build and execute!

as --mno-relax -g -o listing6-9.o listing6-9.s

$ ld --no-relax -o listing6-9 listing6-9.o

$ ./listing6-9

1

4

9

16

25

36

49

64

81

100

121

144

169

196

225

256

289

324

361

400
```


6.3.1 Using the Stack – further examples

This program uses the stack as a buffer to calculate the squares of the first twenty integers. It prints the number to stdout.

Listing 6-10 Using the stack with the squares program

```
.section .data

    newline:    .asciz "\n"
```

```
    .equ begincount,1
    .equ endcount,21
.section .text
.global _start
_start:
    li s0, begincount       # for i = 1 to 20
    li s1, endcount         # if we reach 21 we need to leave!
forloop:
    beq s0, s1, exit  # if i == 21, exit
    mul s2, s0, s0    # Square the number in the counter s2 = i * i
    # Convert and Print Square
    mv a0, s2
    jal ra, print_integer
    # Print Newline
    li a0, 1            # stdout
    la a1, newline
    li a2, 1
    li a7, 64           # sys_write
    ecall
    addi s0, s0, 1      # count++
    j forloop
exit:
    li a0, 0            # Return 0 status
    li a7, 93           # sys_exit
    ecall
# Could have used memory as buffer but using the stack is more educational
print_integer:
    addi sp, sp, - 32   # Allocate 32 bytes on stack
    addi t0, sp, 31     # t0 points to the bottom of the stack
    li t1, 10           # Divisor in decimal system
    # Count the number of characters before returning zero when dividing, convert to ASCII and set
up syscall write parameters in the argument registers
conv_to_ascii:
```

```
    rem t2, a0, t1      # Get digit
    addi t2, t2, 48     # Convert to ASCII
    addi t0, t0, -1     # Move pointer back BEFORE storing
    sb t2, 0(t0)        # Place ascii char in the bottom of the stack
    div a0, a0, t1
    bnez a0, conv_to_ascii

    # Calculate length for sys_write
    # Length = (Initial end of stack) - (current t0):wq
    addi t1, sp, 31
    sub a2, t1, t0      # length in a2
    mv a1, t0           # buffer address in a1
    li a0, 1            # stdout
    li a7, 64           # sys_write
    ecall
    addi sp, sp, 32     # Free up stack
    ret
```

Build and run!

```
$ as -g -o listing6-10.o listing6-10.s
$ ld -o listing6-10 listing6-10.o
$ ./listing6-10
1
4
9
16
25
36
49
64
81
```

```
100
121
144
169
196
225
256
289
324
361
400
```

Re-visit `strace` to view the syscalls

$ strace -c ./listing6-10

```
1
4
9
16
25
36
49
64
81
100
121
144
169
196
225
256
289
```

```
324
361
400
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0        40           write
  0.00    0.000000           0         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000           0        41           total
```

Note: The repeated remainder algorithm that converts from binary to decimal is described in chapter one of the book.

The next program uses an external macro to print a user-defined string. The main program calls the macro code `print_str`.

Listing 6-11 Main program passing a sting to be printed

```
#This is the main program
section .data
    mystring: .ascii "This was printed by a macro\n"
    msglength= 28
.include "print_str.s"
.section .text
.global _start
_start:
# Call the macro  (PRINT_STR_) to print the above text
    PRINT_STR 1, mystring, msglength
exit:
    li a7,93
    ecall
```

Listing 6-12 Macro program to print string

```
# This is the macro program print_str.s
.macro PRINT_STR filedescr, msgaddr, msglength
    # Macro to print a user supplied string
    # Save all argument registers that are being used by the macro
    addi sp,sp,-32
```

```
    sd a0, 24(sp)
    sd a1,16(sp)
    sd a2, 8(sp)
    sd a7,0(sp)
    li a0, \filedescr    # stdout
    la a1, \msgaddr   # User defined string
    li a2, \msglength   # Length of string
    li a7, 64           # sys_write
    ecall
    ld a7,(sp)
    ld a2,8(sp)
    ld a1,16(sp)
    ld a0,24(sp)
    addi sp,sp,32
.endm
```

Build and execute!

```
cp listing6-12.s print_str.s
$ as -g -o listing6-11.o listing6-11.s
$ ld -o listing6-11 listing6-11.o
$ ./listing6-11
This was printed by a macro
```

6.3.1.1 Macros vs Routines

- Modifying the main program to call the macro three times
    - Will result in three copies of the inline code, which can be verified by objdump

Listing 6-13 Calling a macro multiple times

```
#This is the main program
.section .data
    mystring: .ascii "This was printed by a macro\n"
    msglength= 28
.include "print_str.s"
```

```
.section .text
.global _start
_start:
# Call the macro  (PRINT_STR_) to print the above text
    PRINT_STR 1, mystring, msglength
    PRINT_STR 1, mystring, msglength
    PRINT_STR 1, mystring, msglength
exit:
    li a7,93
    ecall
```

Build and run!

```
$ as -g -o listing6-13.o listing6-13.s
$ ld -o listing6-13 listing6-13.o
$ ./listing6-13
This was printed by a macro
This was printed by a macro
This was printed by a macro
```

Disassemble

```
$ objdump -d listing6-13
listing6-13:     file format elf64-littleriscv
Disassembly of section .text:
00000000000100e8 <_start>:
   100e8:     fe010113            addi   sp,sp,-32
   100ec:     00a13c23            sd     a0,24(sp)
   100f0:     00b13823            sd     a1,16(sp)
   100f4:     00c13423            sd     a2,8(sp)
   100f8:     01113023            sd     a7,0(sp)
   100fc:     00100513            li     a0,1
   10100:     00001597            auipc  a1,0x1
   10104:     0b058593            addi   a1,a1,176 # 111b0 <__DATA_BEGIN__>
   10108:     01c00613            li     a2,28
   1010c:     04000893            li     a7,64
```

```
   10110:	00000073            ecall
   10114:	00013883            ld      a7,0(sp)
   10118:	00813603            ld      a2,8(sp)
   1011c:	01013583            ld      a1,16(sp)
   10120:	01813503            ld      a0,24(sp)
   10124:	02010113            addi    sp,sp,32
   10128:	fe010113            addi    sp,sp,-32
   1012c:	00a13c23            sd      a0,24(sp)
   10130:	00b13823            sd      a1,16(sp)
   10134:	00c13423            sd      a2,8(sp)
   10138:	01113023            sd      a7,0(sp)
   1013c:	00100513            li      a0,1
   10140:	00001597            auipc   a1,0x1
   10144:	07058593            addi    a1,a1,112 # 111b0 <__DATA_BEGIN__>
   10148:	01c00613            li      a2,28
   1014c:	04000893            li      a7,64
   10150:	00000073            ecall
   10154:	00013883            ld      a7,0(sp)
   10158:	00813603            ld      a2,8(sp)
   1015c:	01013583            ld      a1,16(sp)
   10160:	01813503            ld      a0,24(sp)
   10164:	02010113            addi    sp,sp,32
   10168:	fe010113            addi    sp,sp,-32
   1016c:	00a13c23            sd      a0,24(sp)
   10170:	00b13823            sd      a1,16(sp)
   10174:	00c13423            sd      a2,8(sp)
   10178:	01113023            sd      a7,0(sp)
   1017c:	00100513            li      a0,1
   10180:	00001597            auipc   a1,0x1
   10184:	03058593            addi    a1,a1,48 # 111b0 <__DATA_BEGIN__>
   10188:	01c00613            li      a2,28
   1018c:	04000893            li      a7,64
```

```
   10190:     00000073            ecall
   10194:     00013883            ld     a7,0(sp)
   10198:     00813603            ld     a2,8(sp)
   1019c:     01013583            ld     a1,16(sp)
   101a0:     01813503            ld     a0,24(sp)
   101a4:     02010113            addi   sp,sp,32
00000000000101a8 <exit>:
   101a8:     05d00893            li     a7,93
   101ac:     00000073            ecall
```

Converting the program to use a function instead results in one copy.

Listing 6-14 Calling a routine three times

```
.section .data
    mystring: .ascii "This was printed by a function\n"
    msglength= 31 #
.include "print_str.s"
.section .text
.global _start
_start:
# Call the function (print_str_) to print the above text
    jal print_str
    jal print_str
    jal print_str
    j exit
print_str:
    # This function is used to print a string
    # Save all argument registers that are being used by the function
    addi sp,sp,-32
    sd a0, 24(sp)
    sd a1,16(sp)
    sd a2, 8(sp)
    sd a7,0(sp)
```

```
    li a0, 1    # stdout
    la a1, mystring   # Message string
    li a2, msglength   # Length of string
    li a7, 64           # sys_write
    ecall
    ld a7,(sp)
    ld a2,8(sp)
    ld a1,16(sp)
    ld a0,24(sp)
    addi sp,sp,32
Ret
exit:
       li a7, 93
       ecall
```

objdump shows –

```
$ objdump -d listing6-14
listing6-14:     file format elf64-littleriscv
Disassembly of section .text:
00000000000100e8 <_start>:
   100e8:    010000ef            jal    100f8 <print_str>
   100ec:    00c000ef            jal    100f8 <print_str>
   100f0:    008000ef            jal    100f8 <print_str>
   100f4:    0480006f            j      1013c <exit>
00000000000100f8 <print_str>:
   100f8:    fe010113            addi   sp,sp,-32
   100fc:    00a13c23            sd     a0,24(sp)
   10100:    00b13823            sd     a1,16(sp)
   10104:    00c13423            sd     a2,8(sp)
   10108:    01113023            sd     a7,0(sp)
   1010c:    00100513            li     a0,1
   10110:    00001597            auipc  a1,0x1
   10114:    03458593            addi   a1,a1,52 # 11144 <__DATA_BEGIN__>
```

```
   10118:     01f00613            li     a2,31
   1011c:     04000893            li     a7,64
   10120:     00000073            ecall
   10124:     00013883            ld     a7,0(sp)
   10128:     00813603            ld     a2,8(sp)
   1012c:     01013583            ld     a1,16(sp)
   10130:     01813503            ld     a0,24(sp)
   10134:     02010113            addi   sp,sp,32
   10138:     00008067            ret
000000000001013c <exit>:
   1013c:     05d00893            li     a7,93
   10140:     00000073            ecall
```

6.3.2 Macros and routines – numeric labels

Some of the programs here use numeric labels such as 1:,2: . . . These are *local labels*. If regular labels are used within the macro, errors will occur during assembly. This is because the label appears multiple times and a global can only occupy a single location, so this cannot be reconciled.

Listing 6-15 Use of numeric labels

```
.macro PRINT_STR filedescr, msgaddr, msglength
    # Macro to print a user supplied string
    # Save all argument registers that are being used by the macro
savestack:
    addi sp,sp,-32
    sd a0, 24(sp)
    sd a1,16(sp)
    sd a2, 8(sp)
    sd a7,0(sp)
    li a0, \filedescr    # stdout
    la a1, \msgaddr   # User defined string
    li a2, \msglength   # Length of string
    li a7, 64           # sys_write
    ecall
restorestack:
```

```
    ld a7,(sp)
    .endm
```

Assembling causes errors –

```
$ as -g -o listing6-13.o listing6-13.s
listing6-13.s: Assembler messages:
listing6-13.s:5: Error: symbol `savestack' is already defined
listing6-13.s:11:  Info: macro invoked from here
listing6-13.s:16: Error: symbol `restorestack' is already defined
listing6-13.s:11:  Info: macro invoked from here
listing6-13.s:5: Error: symbol `savestack' is already defined
listing6-13.s:12:  Info: macro invoked from here
listing6-13.s:16: Error: symbol `restorestack' is already defined
listing6-13.s:12:  Info: macro invoked from here
. . .
```

Numeric labels behave differently.

- They will not encounter *name collisions*.
- Defined by a *single* digit followed by a colon:
- Even though they are defined by a single digit, this is not a constraint, as they can appear multiple times in the same program.
- Use b(ackward) or f(orward) to specify the nearest label.

An example follows:

```
. . .
    sd  s0, 0(sp) # Callee has the burden to save the S registers
    mv   s0, a0         # save pointer to the string which was loaded into a0
    # find string length
    mv   t0, s0
1:
    lbu  t1, 0(t0) # Put current character of string into t1
    beqz t1, 1f  # If t1 equals zero then we have reached the end of the string
    addi t0, t0, 1 # if not get the next character string
    j    1b # Jump backwards to label 1:
1: # OK we now have reached the end of the string as used with .asciz
```

```
    sub  a2, t0, s0     # length is now computed from t0
    mv   a1, s0         # buffer address loaded into a1
    li   a0, 1          # stdout
. . .
```

Key points are:

- Numeric labels are resolved during assembly not during linking
- An instruction such as `bnez t0, 1b` is reconciled to an instruction such as bnez 0x11008
- Consequently, they will not show up as symbols.

Note the disassembly below:

```
Disassembly of section .text:

00000000000100e8 <_start>:
   100e8:       00001517                auipc   a0,0x1
   100ec:       06050513                addi    a0,a0,96 # 11148 <__DATA_BEGIN__>
   100f0:       010000ef                jal     10100 <print_str>
   100f4:       00000513                li      a0,0
   100f8:       05d00893                li      a7,93
   100fc:       00000073                ecall

0000000000010100 <print_str>:
   10100:       ff010113                addi    sp,sp,-16
   10104:       00113423                sd      ra,8(sp)
   10108:       00813023                sd      s0,0(sp)
   1010c:       00050413                mv      s0,a0
   10110:       00040293                mv      t0,s0
   10114:       0002c303                lbu     t1,0(t0)
   10118:       00030663                 beqz    t1,10124 <print_str+0x24>  Note address substituted
for label 1f
   1011c:       00128293                addi    t0,t0,1
   10120:         ff5ff06f                       j         10114 <print_str+0x14>        Note address
substituted for label 1b
   10124:       40828633                sub     a2,t0,s0
   10128:       00040593                mv      a1,s0
   1012c:       00100513                li      a0,1
```

```
  10130:       04000893                li      a7,64
  10134:       00000073                ecall
  10138:       00813083                ld      ra,8(sp)
  1013c:       00013403                ld      s0,0(sp)
  10140:       01010113                addi    sp,sp,16
  10144:       00008067                ret
```

6.3.3 Push and Pop Macros

We have seen that saving values to the stack requires :

- Allocation of space by adjusting the stack pointer
- Saving the registers into the memory location pointed to by the stack with an offset
- Restoring the registers
- Deallocation of stack space by adjusting the stack pointer

Many programmers are more familiar with stack manipulation by using push and pop instructions.

The next two listings show macros that implement single register push and pop instructions.

Listing 6-16 Push Macro

```
.macro PUSH pushregister
        addi sp, sp, -8
        sd   \pushregister, 0(sp)
        .endm
```

Listing 6-17 Pop Macro

```
.macro POP popregister
        ld   \popregister, 0(sp)
        addi sp, sp, 8
        .endm
```

The next listing shows push and pop in action.

- The next example shows the temporary registers being saved and restored across calls:

Listing 6-18 Using the push and pop macros

```
.section .text
.global _start
.include "pushmacro.S"
.include "popmacro.S"
```

```
_start:
    li t0, 10
    li t1, 20
    li t2, 30
    push t0
    push t1
    push t2
    call nonleaf          # call nonleaf function, which calls a leaf function
    pop t2
    pop t1
    pop t0
# exit(result held in register a0)
    li   a7, 93             # sys_exit
    ecall
#---------------------------------------------------------#

# Now in nonleaf function which calls another function → register ra is saved by the push macro
nonleaf:
# overwrite temp registers
    li t0, 110
    li t1, 120
    li t2, 130
    push t0
    push t1
    push t2
    push ra # Save ra
    call leaf               # call leaf function
# Back from nonleaf
    addi a0, a0, 10          # Add 10 to the leaf function's return value
    pop ra # Restore ra
    pop t2
    pop t1
```

```
    pop t0
    ret
# Now in leaf function
leaf:
# Since we are a leaf function we do not need to save ra
    li a0, 42            # return the answer to life, meaning ...
    li t0, 200
    li t1, 300
    li t2, 400
    ret
```

Verify the return value -

```
./nestedfunctions
$ echo $?
52
```

6.3.4 Macros and routines – POP and PUSH Caveats

- The implementation shown is ABI compliant but uses 128 bits to store a single register.
    - With RV64 only 64 bits are needed
- It is possible to use 64-bits for single register allocation on RV64 but it breaks ABI alignment rules.
    - Allocating 128 bits is clean and compliant
- The macros can be easily modified to store two or more registers on the stack.
- The lack of native push and pop instructions, is not an oversight, instead it is part of the design philosophy of RISC-V.
- Since the push and pop macros are made up of multiple instructions, they are not Atomic and could conceivably cause issues with autonomous events.

6.4 Exercises for chapter6

1. What is the purpose of the RA register?
2. Modify the program `maina.s` to keep running after an error message or successful result has been printed by asking the user if they would like to input another value (or not)
3. Why is there an offset of 12 in the instruction `sw a0, 12(sp)?`
4. Explain the difference between a function and a macro.
5. Which directives signify the start and end of a macro?
6. When is the `.include` directive used?
7. Modify one or more of the programs to make better use of functions, and macros. Compare the results using `strace`.
8. Explain why leaf routines do not need to save the RA register.
9. Modify the push and pop macros to function with two registers at a time.
10. Check the program below to find the error. This is a common real-world error that could go unnoticed - If you cannot see the error, then use GDB to trace the register contents.

```
.section .text
.global _start
.include "pushmacro.S"
.include "popmacro.S"
_start:
# Put values in the temp registers to check out macros
    li t0, 10
    li t1, 20
    li t2, 30
    push t0
    push t1
    push t2
    call nonleaf          # call nonleaf function, which calls a leaf function
    pop t2
    pop t1
    pop t0
# exit(result held in register a0)
    li   a7, 93             # sys_exit
    ecall
```

```
# Cannot get here!


# Now in non_leaf function which calls another function → register ra is saved by the push macro
nonleaf:
# overwrite temp registers
    li t0, 100
    li t1, 200
    li t2, 300
    push t0
    push t1
    push t2
    push ra # Save ra
    call leaf               # call leaf function
# Back from non-leaf
    addi a0, a0, 10          # Add 10 to the leaf function's return value
    pop ra # Restore ra
    pop t0
    pop t1
    pop t2
    ret


# Now in leaf function
leaf:
# Since we are a leaf function we do not need to save ra
    li a0, 42            # return the answer to life, meaning ...
    li t0, 2000
    li t1, 3000
    li t2, 4000
    ret
```

6.4.1 Instructions used in chapter 6

Stack Management Instructions

ADDI – Used to adjust the stack pointer (sp)

Example: addi sp, sp, -32 (allocate stack space)

SD – Store doubleword (store register onto stack)

LD– Load doubleword (retrieve register from stack)

Control Transfer / Function Support

JAL – Jump and Link (used to call functions)

RET – Return from function (pseudo-instruction for jalr x0, ra, 0)

JALR– Jump and Link Register (used indirectly via ret)

Macros and Utilities

.macro / .end_macro – Assembler directives for defining macros (not instructions but critical to macro usage)

7 RISC-V assembly and C together

7.1 Overview of the chapter

Chapter 7 explores how assembly language and C code can work together, bridging low-level and high-level programming. It's highly practical for system developers who want to embed performance-critical routines in C-based applications. This chapter will show how to combine RISC-V assembly language with the C programming language. Embedded application developers may initially resort to development using pure machine code, but as we have seen in the earlier chapters, we have the benefit of development under the Linux operating system. There is an advantage to using system calls, especially in the area of screen output and keyboard input. This is best illustrated for those using real hardware, such as the platforms described earlier. An operating system such as Linux allows us to use C code, which can be compiled together with RISC-V assembly.

7.2 Example C code

Consider the basic C program shown below:

Listing 7-1 Basic C program

```
// Addprog.c
# include <stdio.h>
int main() {
int first=10, second=20, sum;
       // Calculate the sum
       sum = first + second;
/      / Display the result
       printf("%d + %d = %d\n", first, second, sum);
return 0;
}
```

Compile and run

```
$ gcc  listing7-1.c
$ ./a.out
10 + 20 = 30
$
```

Compiling generates intermediate files, and by default, the compiler will delete these intermediate files once the executable program has been generated. To retain the generated assembly files, use the `-S` option.

```
$ gcc -S addprog.c
$ ls
addprog.c addprog.s
```

Print the intermediate assembly code.

```
$ cat addprog.s
.option pic
.attribute                                                                    arch,
"rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0_zba1p0_zbb1p0_zbc1p0_zbs1p0"
.attribute unaligned_access, 1
.attribute stack_align, 16
.text
.section .rodata
.align 3
.LC0:
.string "%d + %d = %d\n"
.text
.align 1
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
addi sp,sp,-32
.cfi_def_cfa_offset 32
sd ra,24(sp)
sd s0,16(sp)
cfi_offset 1, -8
.cfi_offset 8, -16
addi s0,sp,32
.cfi_def_cfa 8, 0
```

```
li a5,10
sw a5,-28(s0)
li a5,20
sw a5,-24(s0)
lw a5,-28(s0)
mv a4,a5
lw a5,-24(s0)
addw a5,a4,a5
sw a5,-20(s0)
lw a3,-20(s0)
lw a4,-24(s0)
lw a5,-28(s0)
mv a2,a4
mv a1,a5
lla a0,.LC0
call printf@plt
li a5,0
mv a0,a5
ld ra,24(sp)
.cfi_restore 1
ld s0,16(sp)
.cfi_restore 8
.cfi_def_cfa 2, 32
addi sp,sp,32
.cfi_def_cfa_offset 0
jr ra
.cfi_endproc
.LFE0:
.size main, .-main
```

The stack pointer has been set up with the instruction `addi sp,sp,-32`. You can see how the variables *first* and *second* are assigned in the highlighted instructions `li a5,10` and `li a5, 20`. In between and after the load immediate instructions, there is a stack push to save register a5. A copy of register a5 is loaded into register a4 via the move instruction. Both registers are added together using `addw a5,a4,a5` and the result is stored on the stack.

Following this, the output print parameters are set up via the argument registers a0, a1 and a2.

7.3 Optimizing code with GCC

Table 7- 1 shows the main levels of code optimization that the compiler can generate. The assembly code that was generated by the compiler uses the default optimization level. Generally, optimization level 2 is considered a good compromise[48]. Level 3 can use inline code like macros but can the optimization at the expense of size.

7.4 C optimization techniques

If the file has been compiled with the `-g` option then the optimization level may be included in the executable. The grep utility can be used for this with the dash a switch denoting that the file is an executable file. The extract shows that level 3 optimization was used with this binary.

```
$ grep -a "\-O" a.out

?  ?                  first__fmtlong  unsigned  intunsigned  charmainlong  intshort  unsigned
intprintfsecond__printf_chkshort  intGNU  C17  14.2.0  -mtune=spacemit-x60  -mabi=lp64d  -misa-
spec=20191213  -mtls-dialect=trad  -march=rv64imafdc_zicsr_zifencei_zba_zbb_zbc_zbs  -g  -O3  -
fstack-protector-strongaddprog.c/home/alan/c/usr/include/riscv64-linux-gnu/bitsstdio2.hstdio2-
decl?h
```


[48] *It is recommended to stay with the default levels of optimization until the testing and debugging phases have been carried out.*

Table 7- 1 C optimization levels

Optimization level	Description	Effect
O0	No optimization	Default, easier to debug
O1	Basic optimization	Small optimization, not significantly increasing compilation time
O2	Recommended	Perform optimizations that do not involve space to speed tradeoffs such as inlining bloat – a good compromise, safe!
O3	Aggressive	Uses O2 optimizations and uses inlining for loops; it can slow down compilation
Ofast	Aggressive, disregards strict standards compliance	Uses O3 and optimizations that are not valid for all standard-compliant programs
Oz	Smaller size	Uses O2, excluding optimizations that may increase size

7.4.1 Compile-time optimization

The same program (`addprog.c`) that we saw earlier has been compiled with optimization level 3. You may notice that it does not use the `addw` instruction; instead, it eliminates the variables *first*, *second* and *sum* and calculates the result first. The only reason that the values 10 and 20 are retained is for the print string. The compilation command is:

```
$ gcc –O3 addprog.s
```

The resulting program is much smaller in size. With optimization level 3 the compiler pre-calculates the addition since the values of *first* and *second* are constant and do not change. This eliminates the `addw` instruction. In addition, stack handling has been reduced. It is important to note that optimized code may be harder to debug since it has performed optimizations that may be harder to spot because there is not necessarily a one-to-one correspondence.

7.4.1.1 Constant folding and copy propagation

These techniques are known *as constant folding* and *copy propagation*. Constant folding occurred here by evaluating 10+20 and using a single variable to store the result. In the code shown, it was achieved with the instruction li a4,30. In C terms, it would simply look like sum=30 without using the variables *first* and *second*. Constant propagation will replace the variables with their values, so it can replace the variables *first* and *second* with the constants 10 and 20, so instead of a statement like *sum = first + second,* the compiler can directly use sum = 10+20.

$ cat addprog.s

```
.file "addprog.c"
```

```
.option pic
.attribute                                                                    arch,
"rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0_zba1p0_zbb1p0_zbc1p0_zbs1p0"
.attribute unaligned_access, 1
.attribute stack_align, 16
.text
.section .rodata.str1.8,"aMS",@progbits,1
.align 3
.LC0:
.string "%d + %d = %d\n"
.section .text.startup,"ax",@progbits
.align 1
.globl main
.type main, @function
main:
.LFB23:
.cfi_startproc
addi sp,sp,-16
.cfi_def_cfa_offset 16
li a4,30
li a3,20
li a2,10
lla a1,.LC0
li a0,2
sd ra,8(sp)
.cfi_offset 1, -8
call __printf_chk@plt
ld ra,8(sp)
.cfi_restore 1
li a0,0
addi sp,sp,16
.cfi_def_cfa_offset 0
```

```
jr ra
.cfi_endproc
.LFE23:
.size main, .-main
```

7.4.2 Run-time optimization

Consider the basic C program shown below. This program does not know the variable's values at compile time.

Listing 7-2 C program with user input

```
# include <stdio.h>
int main() { int first, second, sum;
printf("Enter two digit integers: ");
scanf("%d %d", &first, &second);
sum = first + second;
printf("%d + %d = %d\n", first, second, sum);
return 0; }
```

Compile and run

```
$ gcc addprog1.c
$ ./a.out
Enter two integers: 4
3
4 + 3 = 7
$
C
```

The program asks the user for input, so the values of the variables *first* and *second* are not known until runtime. Consequently, constant folding and copy propagation will not apply. Compile time optimization is a static process, unlike run time optimization, which responds to dynamic conditions. Some of the techniques used in run-time optimization include **caching,** which remembers the result of a lookup function, and adaptive inlining, which decides which functions should be inlined by monitoring execution frequency.

7.5 Calling assembly functions from a high-level language

The next example creates two source programs: one written in C[49] code and the other in RISC_V assembly. The C program (Listing 7-3) shown declares an external function (`getproduct`), located in the assembly program (Listing 7-4) and calls it passing the two arguments via a0 and a1. It then calls the `printf` function to output the result. The assembly program is executed in the normal fashion, generating an object file. The `gcc` program[50] generates the C object file and links it with the previously generated object file.

Listing 7-3 C program calling an external assembly routine

```
/* This code shows how to call an assembly language program from C
Listing 7-3.c*/
#include <stdio.h>
// Declare the assembly function
extern int getproduct(int a, int b);
int main() {
int x = 100, y = 200;
// Call RiscV assembly function
int result = getproduct(x, y);
printf("The product of %d and %d is %d\n", x, y, result);
return 0;
}
```

Listing 7-4 RISC-V multiply function called from C

```
$ cat listing7-4.s
.text
.global getproduct
getproduct:
mul a0, a0, a1  # Add the two input registers (a0 and a1) and store in a0
ret             # Return
```

The commands to generate the output file are:

[49] *None of the c code presented here is overly complex, however if the reader is not familiar with C, there is a wealth of on-line tutorials to be consumed that will cover the basics for what is needed here.*

[50] *An alternative c compiler is `clang` which can be installed by `sudo apt install -y clang`.*

```
$ as -g -o listing7-4.o listing7-4.s
$ gcc listing7-3.c listing7-4.o -o outputfile
```

Here `gcc (`GNU complier collection) is used instead of the `ld` command that was previously used to perform the linkage.

The generated assembly files from the `.c` listing can be saved during compilation with the option `-save-temps`. Alternatively, as we saw earlier, to just generate the RISC_V assembly code use the command `gcc -S <filename.c>` which generates `<filename.s>`

The command line is:

```
$ gcc -save-temps listing7-3.c listing7-4.o -o outputfile
$ls -l outputfile*
-rwxrwxr-x 1 alan alan  9584 Jan 28 11:49 outputfile
-rw-rw-r-- 1 alan alan 21460 Jan 28 11:49 outputfile-listing7-3.i
-rw-rw-r-- 1 alan alan  2408 Jan 28 11:49 outputfile-listing7-3.o
-rw-rw-r-- 1 alan alan  1024 Jan 28 11:49 outputfile-listing7-3.s
```

This generates the assembly `.s` file shown above. The bolded and italicized comments have been added to help with explanation and were not part of the `-save-temps` output.

```
.option pic
.attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
.attribute unaligned_access, 0
.attribute stack_align, 16
.text
.section      .rodata
.align 3
.LC0:
.string      "The product of %d and %d is %d\n" # String as defined as part of the C source
.text
.align 1
.global      main
.type  main, @function
main:
addi   sp,sp,-32     # Allocate space on the stack
```

```
sd     ra,24(sp)     # Store a double word (64 bits, our architecture is RV64) from the ra register with an offset of 24 from the stack pointer

sd     s0,16(sp)     # Store a double word from the s0(fp) register with an offset of 16 from the stack pointer

addi   s0,sp,32      # Store stack pointer with an offset of 32, from the current stack pointer

li     a5,100        # First factor

sw     a5,-20(s0)    # Store first factor

li     a5,200        # Second factor

sw     a5,-24(s0)    # Both factors stored in addresses pointed to by s0 (s0 -20, s0 024)

lw     a4,-24(s0)    # Load a4 with second factor

lw     a5,-20(s0)    # Load a5 with first factor

mv     a1,a4         # Move second factor to a1

mv     a0,a5         # Move first factor to a0

call   getproduct@plt # Call function with parameters held in a0 and a1

mv     a5,a0         # Move first factor into a5

sw     a5,-28(s0)    # Store first factor into location pointed to by s0 with an offset of -28

lw     a3,-28(s0)    # Load first factor into a3

lw     a4,-24(s0)    # Load second factor into a4

lw     a5,-20(s0)    # Load first factor into a5

mv     a2,a4         # Load second factor into a2

mv     a1,a5         # Load first factor into a1

lla    a0,.LC0

call   printf@plt    # call printf function using the procedure linkage table[51]

li     a5,0

mv     a0,a5

ld     ra,24(sp)     # Pop ra register back to original value

ld     s0,16(sp)     # Pop frame pointer back to original value

addi   sp,sp,32      # Set stack pointer back to original value

jr     ra

. . .
```

[51] *Refer to RISC-V ABIs Specification (https://lists.riscv.org/g/tech-psabi/attachment/61/0/riscv-abi.pdf) section 8.5.6 for more information on the procedure linkage table.*

Looking at the listing, it appears that a great deal of optimization could be performed.

```
.file   "listing7-2.c"
.option pic
.attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
.attribute unaligned_access, 0
.attribute stack_align, 16
.text
.section        .rodata.str1.8,"aMS",@progbits,1
.align  3
.LC0:
.string "Result of adding %d to %d is: %d\n"
.section        .text.startup,"ax",@progbits
.align  1
.globl  main
.type   main, @function
main:
addi    sp,sp,-16
sd      ra,8(sp)
li      a3,15
li      a5,27
#APP
# 8 "listing7-2.c" 1
add a3, a3, a5
# 0 "" 2
#NO_APP
li      a2,27
sext.w  a3,a3
li      a1,15
lla     a0,.LC0
call    printf@plt
ld      ra,8(sp)
```

```
li      a0,0
addi    sp,sp,16
jr      ra
.size   main, .-main
.ident  "GCC: (Debian 12.2.0-14) 12.2.0"
.section        .note.GNU-stack,"",@progbits
```

The optimization options are shown in Table 7- 1

An optimized listing is shown below:

```
. cat outputfileOz-listing7-3.s
        .file   "listing7-3.c"
        .option pic
        .attribute                                                                  arch,
"rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0_zba1p0_zbb1p0_zbc1p0_zbs1p0"
        .attribute unaligned_access, 1
        .attribute stack_align, 16
        .text
        .section        .rodata.str1.8,"aMS",@progbits,1
        .align  3
.LC0:
        .string "The product of %d and %d is %d\n"
        .section        .text.startup,"ax",@progbits
        .align  1
        .globl  main
        .type   main, @function
main:
.LFB13:
        .cfi_startproc
        addi    sp,sp,-16
        .cfi_def_cfa_offset 16
        li      a1,200
        li      a0,100
```

```
        sd      ra,8(sp)
        .cfi_offset 1, -8
        call    getproduct@plt
        mv      a4,a0
        li      a3,200
        li      a2,100
        lla     a1,.LC0
        li      a0,2
        call    __printf_chk@plt
        ld      ra,8(sp)
        .cfi_restore 1
        li      a0,0
        addi    sp,sp,16
        .cfi_def_cfa_offset 0
        jr      ra
        .cfi_endproc
.LFE13:
        .size   main, .-main
        .ident  "GCC: (Bianbu 14.2.0-19ubuntu2bb3) 14.2.0"
        .section        .note.GNU-stack,"",@progbits
```

```
$ ls -l outputfileO0-listing7-3.s outputfileOz-listing7-3.s
-rw-rw-r-- 1 alan alan 1024 Jan 28 11:58 outputfileO0-listing7-3.s
-rw-rw-r-- 1 alan alan  830 Jan 28 12:00 outputfileOz-listing7-3.s
```

7.5.1 Clang compiler

The `clang` compiler uses similar optimization options. A comparison of the assembly file size using the `wc` utility[52] is shown following:

```
$ clang with -O0
52  182 1463
```

[52] *`wc` counts lines, words and bytes so an output of 52 182 1463 refers to 52 lines, 182 words and 1463 bytes; using `wc -l` will only return the line count*

```
$ clang with -Oz

39  124 1015 Using inline code
```

The next program uses inline code to execute assembly language instructions from a single C source program. The GNU assembler keyword `ASM` is used to denote the operands using C syntax.

There are two forms of ASM– *Basic* and *Extended*. In-line assembly code is a bridge for interfacing the high-level convenience of C/C++ to the low-level functionality of RISC-V assembly code.

7.5.2 Basic ASM

Basic ASM is a set of assembly instructions. With inline code, the `asm` keyword is not an actual C keyword[53] but it is understood by the assembler. Note that non-GNU assemblers may use an alternative keyword. Basic ASM is simpler than extended ASM and can be used when no operands are involved. The next listing shows an example of in-line Basic ASM used with C code; in practice, Extended ASM is used more often with in-line assembly code.

Listing 7-5 Basic ASM example

```
# include <stdio.h>

const char message[] = "Hello - RiscV Basic ASM!\n";

int main()

{

asm(

"la a0, message\n"      // load address of msg into a0 (1st argument to puts)

"call puts\n"           // call puts(message)

"li a7, 93\n"

"ecall\n"

);

return 0;

}
```

As an exercise, you may wish to run the program with optimization and note the most significant changes.

1.1.1 Extended ASM

Extended ASM can use variables from the C/C++ source code. Extended ASM cannot be used outside of these functions. The assembler template consists of:

```
asm(code template :  output operand(s) : input operand(s) : clobber list);
```

Table 7-1 provides an explanation.

[53] *This is not the case with C++.*

Table 7-1 Inline assembly template

Template		
	Example	Description
Code - Assembler Instruction	`"mul %0, %1, %2"`	Regular assembly instruction
Code Template parameters	`add%[inputa], %[inputb]`	Using parameters passed as inputs to the code template
Output Operand(S) List	`: "=r" (result)` `Can be left empty using :`	List of output operand(s) [answer] is a symbolic name, r is a constraint string meaning register and (result) is returned to the Calling code.
Input Operands List	`[inputa] "r" (a), [inputb] "r" (b)`	Similar syntax to operand list
Clobber List	`"t0", "t1"`	Optional list of registers, that *may* not be preserved

An example of inline assembly code in a C program using Extended ASM is shown following:

Listing 7-6 Extended ASM example

```
cat listing7-6.c
#include <stdio.h>
int main() {
int number1 = 15, number2 = 27, result;
// Using extended ASM to add a and b
asm volatile (
"add %0, %1, %2"
: "=r"(result)  // Output operand
: "r"(number1), "r"(number2)  // Input operands
:                // No clobbered registers
);
printf("Result of adding %d to %d is: %d\n", number1,number2,result);
return 0;
}
```

- This instruction adds two registers indicated by the input operands %1 and %2,
- %0 represents the output operand.

- "=r" indicates that the result (sum) will be stored in a register.
- "r"(number1), "r"(number2) indicates registers.

The intermediate assembly file generated by the C compiler is shown below:

Note the comments are not generated by gcc but edited in for clarity.

```
.option pic
.attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
.attribute unaligned_access, 0
.attribute stack_align, 16
.text
.section        .rodata
.align  3
.LC0:
.string "Result of adding %d to %d is: %d\n"
.text
.align  1
.globl  main
.type   main, @function
main:
addi    sp,sp,-32
sd      ra,24(sp)
sd      s0,16(sp)
addi    s0,sp,32
li      a5,15 //number1 is stored in register a5
sw      a5,-20(s0) // It is then pushed onto the stack
li      a5,27 // number2 is stored in register a5
sw      a5,-24(s0) // It is then stored onto the next stack location
lw      a5,-20(s0) // number1 is retrieved from the stack and stored in register a5
lw      a4,-24(s0) //number2 is retrieved from the stack and stored in register a4
#APP
```

```
add a5, a5, a4 // Number1 is added to number2 storing result in register a5
# 0 "" 2
#NO_APP
sw      a5,-28(s0) // Result is stored onto the stack
lw      a3,-28(s0) // Result is popped form the stack and loaded into register a3
lw      a4,-24(s0) // Restore number2 into register a4
lw      a5,-20(s0) // Restore number1 into register a5
mv      a2,a4 // Store numbers into registers a1 and a2
mv      a1,a5
lla     a0,.LC0     // Set up print output
call    printf@plt
li      a5,0
mv      a0,a5
ld      ra,24(sp)
ld      s0,16(sp)
addi    sp,sp,32
jr      ra
.size   main, .-main
.ident  "GCC: (Debian 12.2.0-14) 12.2.0"
.section        .note.GNU-stack,"",@progbits
```

7.5.2.1 Further Basic ASM example

The C program presented here shows another example of Basic ASM. The C variables are matched to the RISC-V registers t0 through t2 using the register keyword. The addition is performed with the instruction add t2, t0, t1 enclosed within the `asm` block. On entry, it is actually in the text section. After this, the `.rodata section` is defined, and then the last part of the `asm` block is to return to the text section. This is important as the program needs to exit the data section. The printing is performed by the C code. The alternative is to save the section state and then restore it as shown in the listing below.

Listing 7-7 Further BASIC asm example

```
# include <stdio.h>
//Declare the assembly message as an external variable since it is defined in assembly block
extern char assembly_msg[];
int main() {
```

```
    int number1 = 42;
    int number2 = 569;
    int result; //Assign registers to C variables
    register int num1 asm("t0") = number1;
    register int num2 asm("t1") = number2;
    register int output asm("t2");
asm (
"add t2, t0, t1 \n\t"
".section .rodata\n\t"
".global assembly_msg\n\t"
"   assembly_msg:\n\t"
"   .asciz \"The result of adding 42 and 569 (calculated using basic ASM) is: \"\n\t"
".section .text\n\t" // Now back in the text section
);
// Get  the result from the register
    result = output;
    printf("%s %d\n", assembly_msg, result);
    return 0;
}
./a.out
The result of adding 42 and 569 (calculated using basic ASM) is:  611
```

7.6 Format Specifiers

Earlier programs in this chapter written in C have already used `printf` to output results. The C standard library function `printf` is defined within `<stdio.h>` as `int printf(const char *format,…)` It is a *variadic* function, which means that it can take a variable number of arguments. This is conveyed by the ellipsis … in the prototype. The function takes a minimum of one argument, which is a pointer to the location of the starting character of the text.

The text itself can embed formatting tags which specify how the arguments that are passed are to be printed – for example, a variable using "%d" will be formatted as a signed base 10 integer.

The C `printf` function is part of the C standard library. The arguments. are handled according to the ABI rules, with registers a0 through a7 providing the first eight arguments Any remainders are pushed onto the stack. Prior to the first `printf` call, register a0 holds the address of the string. When it parses the string, the routine may or may not encounter one or more format specifiers; if it does, it substitutes the value of register a1 into the string using the

format specified, and it resumes printing until it reaches the end of the string (signified by null) or another format specifier. If there are more format specifiers than argument registers, they will be passed on the stack.

A non-exhaustive list of format specifiers is shown in Table 7-2.

Table 7-2 printf format specifiers

Format specifier	interpretation
%d	Signed decimal number.
%u	Unsigned decimal number.
%s	Pointer to an array of characters.
%c	Outputs a single character.
%x	Represents an unsigned integer in lower case hexadecimal form.
%X	Represents an unsigned integer in upper case hexadecimal form.
%%	Outputs a literal "%" character.
%e	Represents floating-point as decimal exponent notation.
%f	Represents floating-point as decimal.

Using the `printf` specifiers helps immensely when using assembly code, although it should be noted that some systems will not have `printf` available[54]. Listing 7-8 shows examples. Here the registers a0 through a3 are used as function parameters to `printf` as specified in the ABI calling convention. The use of `printf` could conceivably slow down execution in time-dependent code whereas the direct assembly printing methods are faster. The `printf` function expects the first parameter to be a null-terminated string, which uses the `.asciz` assembler directive.

Listing 7-8 Using the printf function with assembly code

```
.extern printf
    .section .rodata
string1:    .asciz "The square of decimal number 42 = %d (Base 10)\n"
string2:    .asciz "The square of number  42 decimal = %x (Base 16)\n"
string3:    .asciz "The square of number 42 decimal =%X (Upper Case hex)\n"
.equ number1,42
    .section .text
    .global main
```

[54] *Typically, this would apply to bare-metal embedded implementations with limited resources.*

```
main:
    #Prologue
    addi sp, sp, -16        # allocate stack (16-byte aligned)
    sd   ra, 8(sp)          # save return address
    li a1, number1
    mul a1,a1,a1

    la   a0, string1        # a0 = pointer to format string
    call printf             # call printf

    li a1, number1
    mul a1,a1,a1
    la a0, string2
    call printf

    li a1, number1
    mul a1,a1,a1
    la a0, string3
    call printf

    # Epilogue
    ld   ra, 8(sp)          # restore return address
    addi sp, sp, 16         # restore stack
    li   a0, 0              # return 0 from main
    ret
```

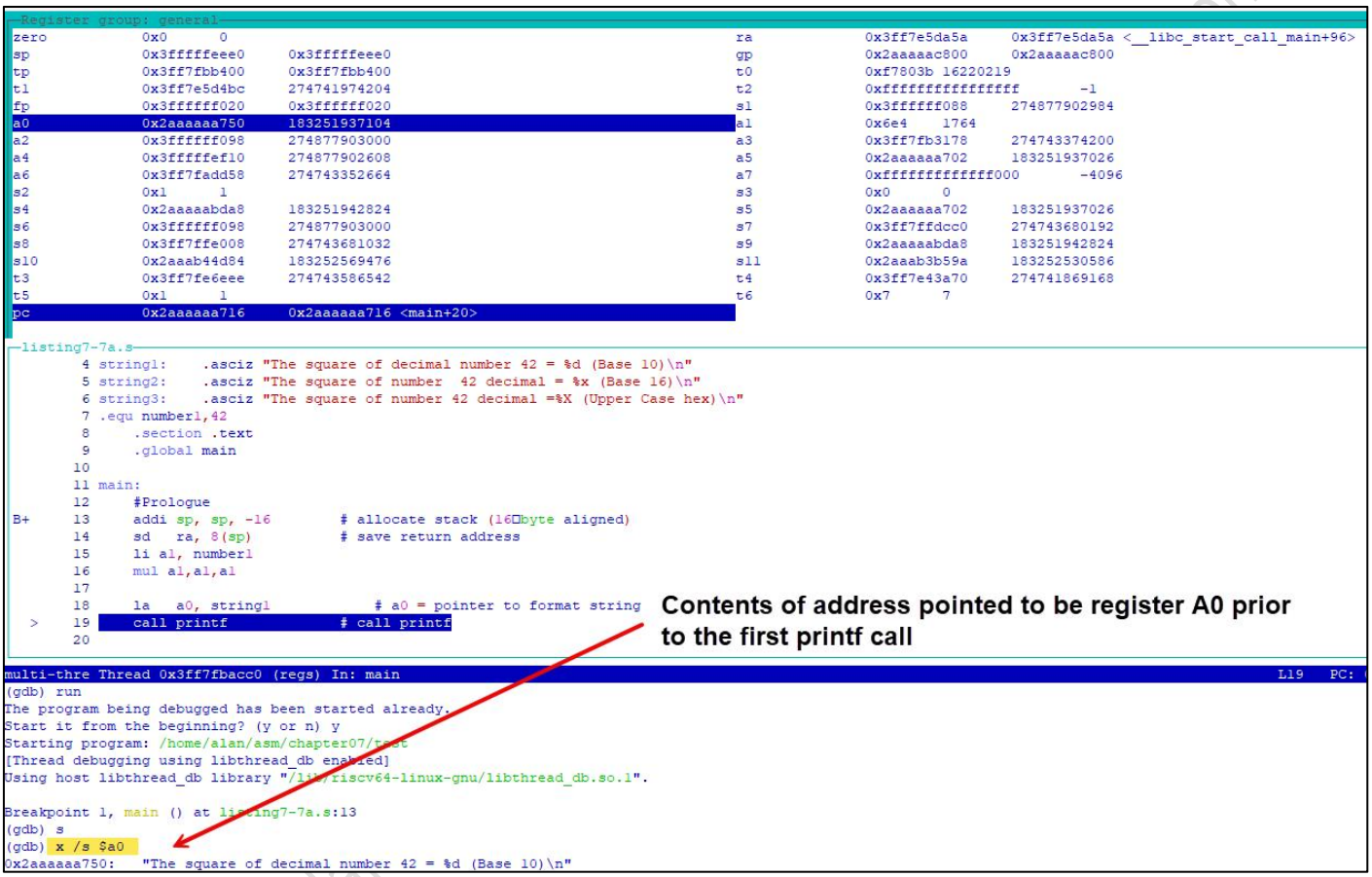
The program was built with `gcc`. Using `gcc` ensures that the program is linked with the C standard library `(libc)` which is necessary for invoking `printf`.

```
gcc -o test listing7-7a.s
./test

The square of decimal number 42 = 1764 (Base 10)
```

```
The square of number 42 decimal = 6e4 (Base 16)

The square of number 42 decimal =6E4 (Upper Case hex)
```

We can see that using `printf` is much simpler than printing with pure assembly. Building with the `-g` option lets us run with GDB.

Figure 7-1 Using GDB with GCC



7.7 Exercises for chapter 7

1. Write a program that could benefit from optimization; include redundant instructions to see how it is handled by the disassembled code.
2. Generate compute-intensive code and see if optimization can reduce runtime.
3. List the parameters and their locations as expected by `printf`.

7.8 RISC-V related instructions used in chapter 7

Inline ASM Tools

- While these are not actual RISC-V instructions, the chapter discusses **basic and extended inline assembly syntax** in GCC using:
  - asm("...") or __asm__ volatile ("...") blocks
  - **Constraints** like r, m, =r, 0, etc.
  - Clobber lists and output/input operands

8 Floating-Point

Overview of the chapter

Chapter 8 introduces floating-point operations in RISC-V, based on the IEEE 754 standard. It explains how floating-point numbers are represented, manipulated, and evaluated in assembly, highlighting both single and double precision.

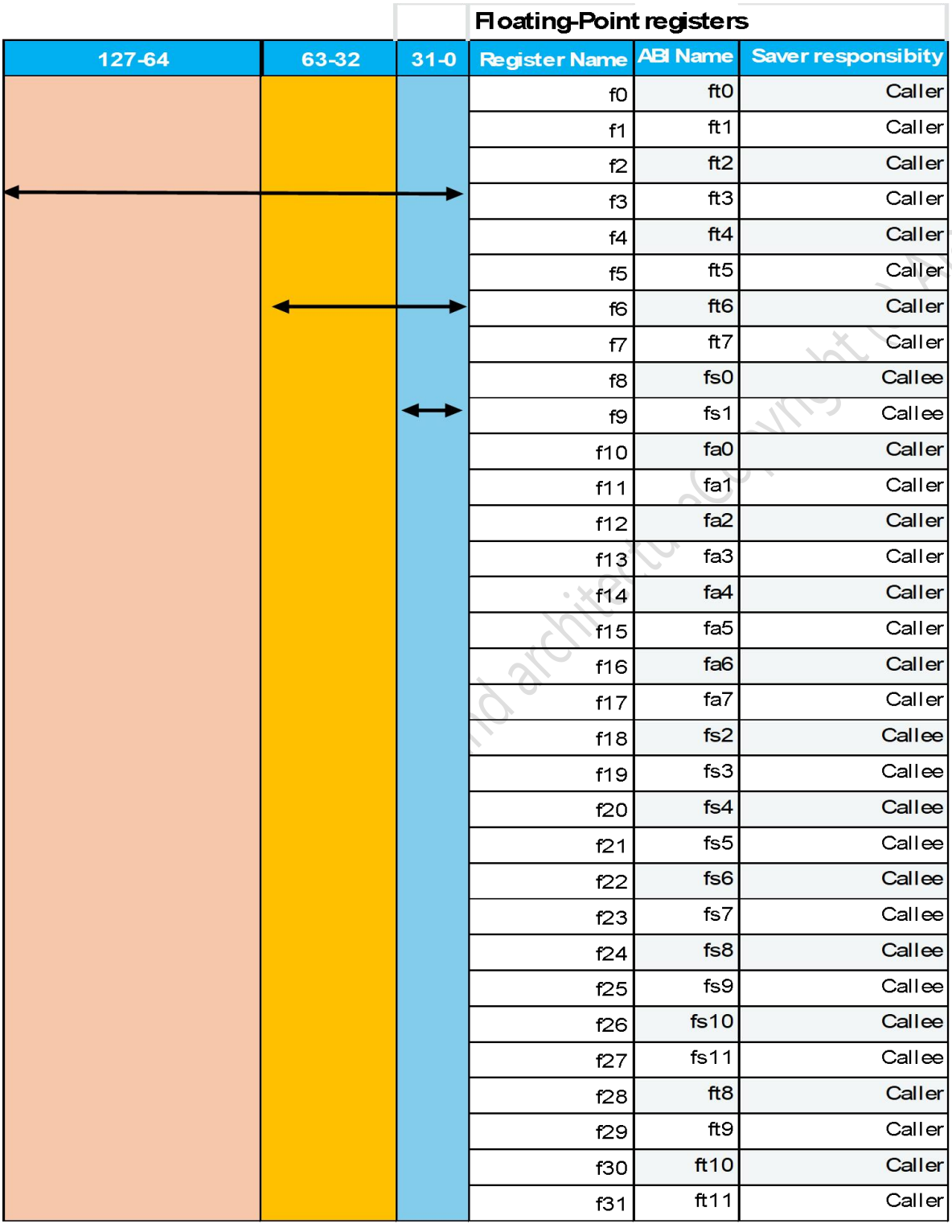
8.1 RISC-V Floating-Point Capability

Not all RISC-V systems can handle floating-point; recall that only some RISC-V systems have floating-point support, as evidenced by their identification string (the F extension), as discussed in chapter 3

8.1.1 Floating-point register set

Capable systems have 32 (f0 → f31) floating-point registers, shown in Figure 8-1, while the register width (FLEN) is determined by the RV extension shown in Table 8-2.

Figure 8-1 Floating-point registers



Floating-Point registers

127-64	63-32	31-0	Register Name	ABI Name	Saver responsibity
			f0	ft0	Caller
			f1	ft1	Caller
			f2	ft2	Caller
			f3	ft3	Caller
			f4	ft4	Caller
			f5	ft5	Caller
			f6	ft6	Caller
			f7	ft7	Caller
			f8	fs0	Callee
			f9	fs1	Callee
			f10	fa0	Caller
			f11	fa1	Caller
			f12	fa2	Caller
			f13	fa3	Caller
			f14	fa4	Caller
			f15	fa5	Caller
			f16	fa6	Caller
			f17	fa7	Caller
			f18	fs2	Callee
			f19	fs3	Callee
			f20	fs4	Callee
			f21	fs5	Callee
			f22	fs6	Callee
			f23	fs7	Callee
			f24	fs8	Callee
			f25	fs9	Callee
			f26	fs10	Callee
			f27	fs11	Callee
			f28	ft8	Caller
			f29	ft9	Caller
			f30	ft10	Caller
			f31	ft11	Caller

Bit 127 Bit 0

32 floating-point registers, data width is determined by RV extension

Table 8-1 indicates the number of bits that are used by floating-point numbers in the RISC-V architecture, for reference a single precision number is referenced as a *float* in the C language and a double precision number as a *double*.

This chapter will only discuss single and double-precision numbers, not half or quad.

Recall from Chapter One:

- Single precision numbers are divided into three fields: a single sign bit, eight bits for a biased exponent and 23 bits for the significand.
- Double precision numbers are divided into three fields: a single sign bit, eleven bits for a biased exponent and 52 bits for the significand.
- With normalized numbers, the leading 1.XXX… is implicit and not coded.

Table 8-1 Bit fields of single and double precision floating-point numbers

Format	Bits	Significand	Unbiased Exponent	Decimal Precision
Single	32	24 (23+1)	8	6-9 digits
Double	64	53 (52+1)	11	15-17 digits

Table 8-2 Floating-point register width

Optional extension	Register width
H Half precision	16 bits (FLEN)
F Single precision	32 bits (FLEN)
D Double precision	64 bits (FLEN)
Q Quad precision	128 bits (FLEN)

8.2 Instruction types

Floating-point instructions can be broadly categorized into the following areas.

8.2.1 Arithmetic instructions

Floating-point arithmetic operations include –

- Add
- Subtract
- Multiply
- Divide
- Square root
- Minimum

- Maximum

8.2.2 Load and store instructions

- Load
- Store

8.2.3 Convert instructions

- Convert from float to unsigned integer
- Convert from unsigned integer to float
- Convert from single precision float to double precision float
- Convert from double precision float to single precision float

8.2.4 Categorization instructions

These are used to ascertain the type of value, such as minus infinity -∞, -0, NaN, . . . The `fclass` instructions are used to store a value corresponding to the type in a destination.

8.2.5 Comparison instructions

This covers the usual comparisons – less than or equal, equal,

8.2.6 Miscellaneous instructions

- Sign-injection which copies from a source to a destination with sign-bit manipulation.

8.3 Instruction format

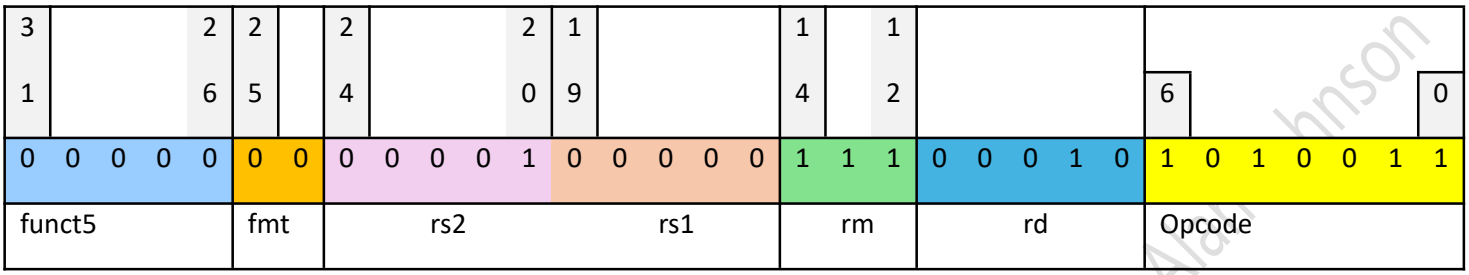
The format of a floating-point instruction is `F<instruction>.<precision> rd, rs1, rs2` where <instruction> is an operation such as <`ADD`>, <`MUL`> or <`DIV`> and <precision> is the floating-point precision such as <S> or <D>, so the instruction `FSUB.S` refers to a single precision floating-point subtraction operation. An arithmetic instruction – `FADD.S f0,f1,f2` will add the contents of registers f1 and f2 placing the result in register f0. The field breakdown of this instruction is as follows:

Table 8-3 Field meaning of FADD.s instruction

Field	Value	Notes
Opcode	1010011 (53)	Used with funct5 to determine operation
rd	00010 (2)	Destination register (F2)
rm	111 (7)	Select the dynamic rounding mode held in frm (rounding mode field) which is the default mode if not specified in the instruction
rs1	00000	First source register (F2)
rs2	00001	Second source register (F1)
fmt	00	S (32-bit) Single precision see
funct5	00000	FADD instruction

Table 8-4 fadd bit fields

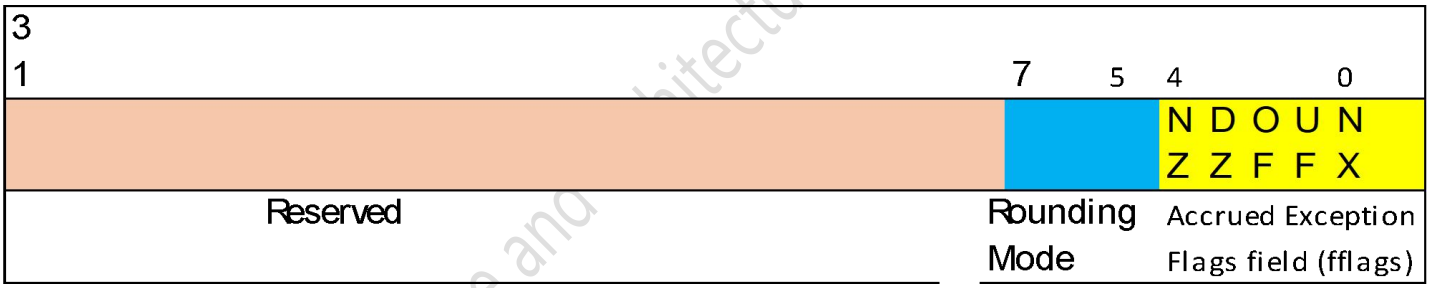
Instruction - 00107153 fadd.s ft2,ft0,ft1



8.3.1 Floating-point control and Status Register

The floating-point control and status register (**FCSR**) is a 32-bit register used to flag exceptions and the *rounding mode* used with floating-point operations. The exception flags occupy bits 4:0 and the rounding mode occupies bits 7:5 as shown.

*Figure 8-2 **FCSR** bit definitions*

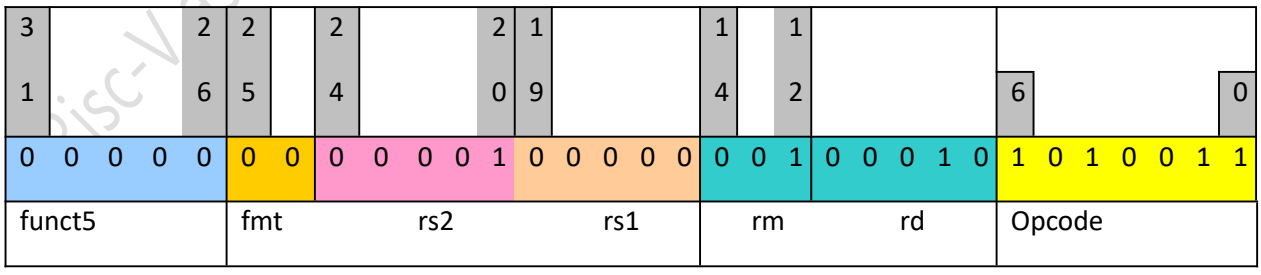


8.3.2 Rounding Modes

There are two types of rounding modes - *dynamic* and *static*. Static rounding modes are specified in the floating-point instruction, such as `fadd.s ft2, ft0. ft1, rtz` where `rtz` stands for *round towards zero*. Static rounding modes are fixed.

The bit-field breakdown for `fadd.s ft2, ft0. ft1, rtz` is shown in Figure 8-4

Table 8-5 Field breakdown of FADD.s f2,f0,f, rtz instruction



Dynamic rounding modes can be changed during code execution; the current rounding mode is specified within the **fcsr** register.

Table 8-6 defines the various rounding modes:

Table 8-6 Rounding mode bits

Mode	Mnemonic	Notes
000	RNE	Round to nearest (even values are preferred)
001	RTZ	Round towards zero
010	RDN	Round down towards -infinity
011	RUP	Round up towards +infinity
100	RMM	Round to nearest (max magnitude)
101	Reserved	
110	Reserved	
111	DYN	Dynamic rounding

8.3.3 Accrued Exception bits

The meaning of the exception bits are:

- NZ Invalid
- OF Overflow
- UF Underflow
- DZ Divide by zero
- NX Inexact

The `frcr rd` command can be used to read the register, placing the result into a general-purpose (integer) register, and the `fscsr rs1` instruction is used to set bits from a source general-purpose register.

Note that the accrued exception bits must be cleared by the software once they have been set!

The first listing in this section adds two double precision floating-point numbers and uses printf[55] to print the result. The address of the numbers pi and e are first placed in the integer registers a0 and a1. They are then loaded into

[55] *Use double-precision with printf. See assembly - How to print a single-precision float with printf - Stack Overflow (https://stackoverflow.com/questions/37082784/how-to-print-a-single-precision-float-with-printf) for elaboration.*

floating-point registers fa0 and fa1. They are added together, placing the result in fa2 by the `fadd.d` instruction. After this, the floating-point values are placed back into the integer registers so that they can be displayed using `printf`.

Listing 8-1 Adding two double-precision floating-point numbers

```
# Double-precision floating-point addition example
.data
pi:   .double 3.141592653589793   # First double-precision number
euler:   .double 2.718281828459045   # Second double-precision number
displayresult: .string "Pi %.15f added to e %.15f = %.15f\n"  # Format string for printf
.text
.global main
main:
        # Load double-precision floating-point numbers
        la a0, pi             # Load address of pi
        fld fa0, 0(a0)          # Load pi value into fa0
        la a0, euler             # Load address of e
        fld fa1, 0(a0)          # Load e value into fa1
        fadd.d fa2, fa0, fa1   # Double-precision addition, result in fa2
        # set printf arguments
        # Move Floating-point numbers into integer registers
        fmv.x.d a1, fa0        # pi goes to a1
        fmv.x.d a2, fa1        # e goes to a2
        fmv.x.d a3, fa2        # Result to a3
        # Load string
        la a0, displayresult   # printf a0 for string, a1,a2,... for other parameters
        # Print the result
        call printf
        # exit
        li a7, 93
        ecall
```

The compilation string used was:

```
$ gcc -g listing8-1.s -o listing8-1
```

And the output shows:

```
./listing8-1

Pi 3.141592653589793 added to e 2.718281828459045 = 5.859874482048838
```

After the floating-point registers have been added their contents are shown:

```
fa0            {float = 3.37028055e+12, double = 3.1415926535897931}    (raw 0x400921fb54442d18)
fa1            {float = -2.85695233e-32, double = 2.7182818284590451}   (raw 0x4005bf0a8b145769)
fa2            {float = -1.06623026e+29, double = 5.8598744820488378}   (raw 0x40177082efac4240)
```

After the floating-point values have been moved back into the integer registers, their contents are:

```
a0             0x2aaaaac010         183251943440
a1             0x400921fb54442d18            4614256656552045848
a2             0x4005bf0a8b145769            4613303445314885481
a3             0x40177082efac4240            4618283650560836160
```

To verify:

Convert 40177082efac4240 to binary.

6666555555555544444444443333333333222222222211111111110000000000
3210987654321098765432109876543210987654321098765432109876543210
01000000000101110111000010000010111 01111010110001000010010000000

- Extract the sign bit (bit 63) = 0 = Positive.
- Extract the exponent field (bits 62:52) = 1025 decimal; the double precision range is -1022 → +1023, biased exponent is 1025-1023 = 2
- Extract the significand field bits 51:0), adding an explicit leading 1 to get:

1.0111011100001000001011101111101011000100001001000000

$= (1 \times 2^{0}) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4}) + (0 \times 2^{-5}) + (1 \times 2^{-6}) + (1 \times 2^{-7}) + (1 \times 2^{-8}) + (0 \times 2^{-9})$

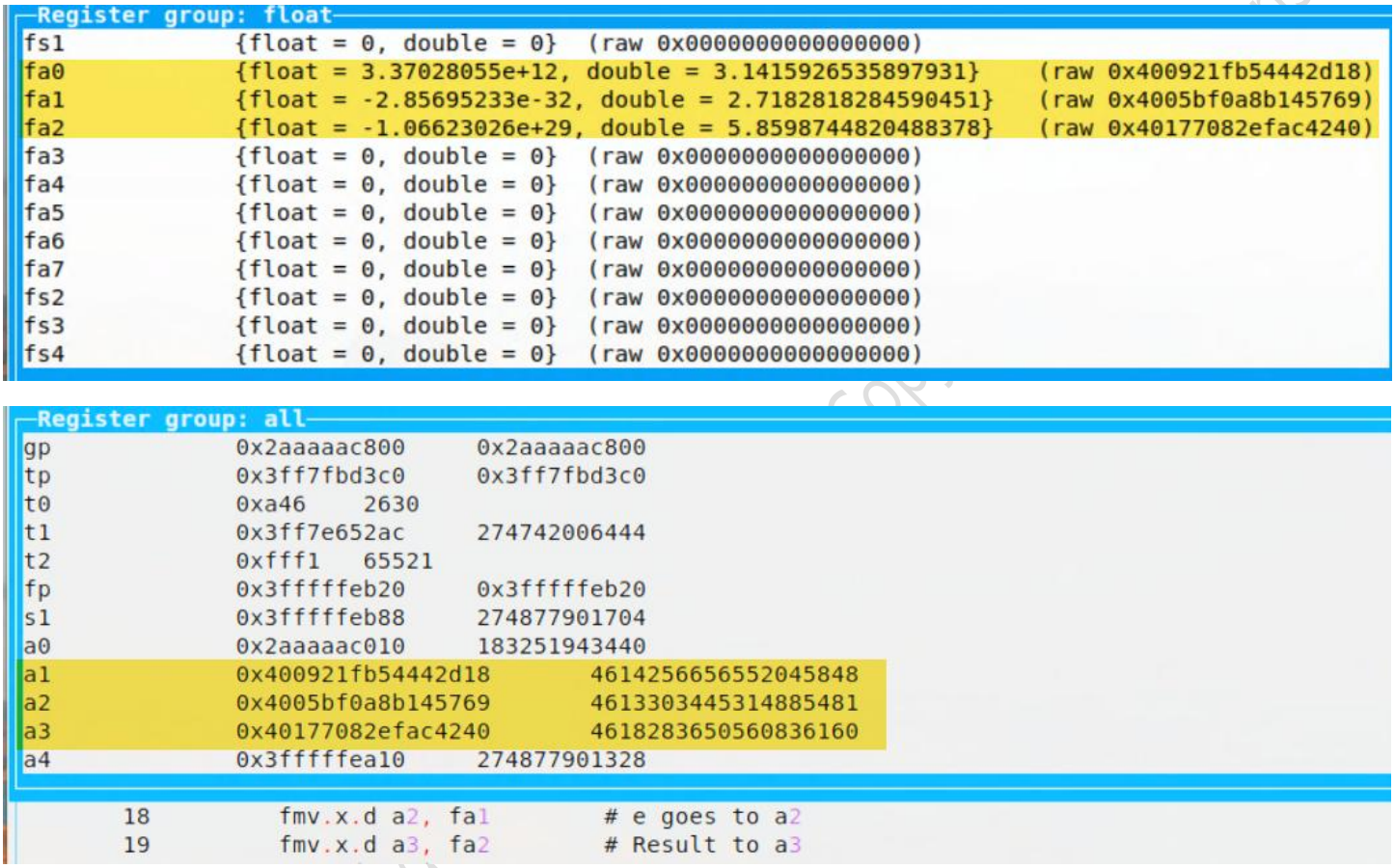
$+ (0 \times 2^{-10}) + (0 \times 2^{-11}) + (0 \times 2^{-12}) + (1 \times 2^{-13}) + (0 \times 2^{-14}) + (0 \times 2^{-15}) + (0 \times 2^{-16}) + (0 \times 2^{-17}) + (0 \times 2^{-18}) + (1 \times 2^{-19})$
$+ (0 \times 2^{-20}) + (1 \times 2^{-21}) + (1 \times 2^{-22}) + (1 \times 2^{-23}) + (0 \times 2^{-24}) + (1 \times 2^{-25}) + (1 \times 2^{-26}) + (1 \times 2^{-27}) + (1 \times 2^{-28}) + (1 \times 2^{-29})$
$+ (0 \times 2^{-30}) + (1 \times 2^{-31}) + (0 \times 2^{-32}) + (1 \times 2^{-33}) + (1 \times 2^{-34}) + (0 \times 2^{-35}) + (0 \times 2^{-36}) + (0 \times 2^{-37}) + (1 \times 2^{-38}) + (0 \times 2^{-39})$
$+ (0 \times 2^{-40}) + (0 \times 2^{-41}) + (0 \times 2^{-42}) + (1 \times 2^{-43}) + (0 \times 2^{-44}) + (0 \times 2^{-45}) + (1 \times 2^{-46}) + (0 \times 2^{-47}) + (0 \times 2^{-48}) + (0 \times 2^{-49})$
$+ (0 \times 2^{-50}) + (0 \times 2^{-51}) + (0 \times 2^{-52})$
$= (1.46496862051220944068)_{10}$

- Multiply by the exponent (obtained earlier) = 1.46496862051220944068 * $(1*2^{2})$ = ~5.86

8.3.4 Viewing the floating-point registers in the debugger:

Use the command `tui reg float` to enable viewing of the floating-point registers with GDB. If not using the TUI, then `info all-registers` will list them.

Figure 8-3 Viewing the FP registers in the GDB TUI



The next section of code introduces the floating-point multiply and divide instructions, along with integer conversion with different types of rounding. Two numbers are multiplied together and then this result is divided by one of the original numbers to see if there are any errors due to precision.

Listing 8-2 Floating-point rounding using static modes

```
.data

number1:                .double 123.141592653589793   # First double-precision number

number2:                .double 422.718281828459045   # Second double-precision number

displaymresult:           .asciz "\n The result of %.15f multiplied by %.15f = %.15f\n"   # Format
string for printf

displaydresult:         .asciz "\n The result of %.15f divided by %.15f = %.15f\n"
```

```
displayrne:             .asciz "\n The integer result rounded to nearest (ties to even) is %d\n"
displayrup:             .asciz "\n The integer result rounded up is %d\n"
displayrdn:             .asciz "\n The integer result rounded down is %d\n"
displayrmm:             .asciz "\n The integer result rounded to nearest (max magnitude) is %d\n"
.text
.global main
main:
   # Load double-precision floating-point numbers
   la a0, number1         # Load address of first number
   fld fa0, 0(a0)         # Load number1 value into fa0
   la a0, number2         # Load address of second number
    fld fa1, 0(a0)         # Load number2 value into fa1
    fmul.d fa2, fa0, fa1   # Double-precision multiplication, result in fa2
    fdiv.d fa3, fa2, fa0   # Now divide (number1*number2)by number1 result in fa3
        fcvt.lu.d t0,fa2,rne
        fcvt.lu.d t1,fa2,rup
        fcvt.lu.d t2,fa2,rdn
        fcvt.lu.d t3,fa2,rmm
        # Set up stack space and push registers t0-t3
        addi sp,sp,-48 #Allocate space
        sd t0, 8(sp)
        sd t1,16(sp)
        sd t2,24(sp)
        sd t3,32(sp)
        # set printf arguments for fa2 value
        # Move Floating-point numbers into integer registers
        fmv.x.d a1, fa0        # number1 goes to a1
        fmv.x.d a2, fa1        # number2 goes to a2
        fmv.x.d a3, fa2        # Result to a3
        # Load multiplication string
        la a0, displaymresult   # printf a0 for string, a1,a2,... for other parameters
```

```
        call printf
        # Print the division result
        la a0, number1
        fld fa0, 0(a0)
        fmv.x.d a1, fa2 #  multiplication result goes into parameter1
        fmv.x.d a2, fa0 # Number1 is parameter2
        fmv.x.d a3, fa3 # derived number2 is parameter3

        # Load division string
        la a0, displaydresult
        call printf
        # Now show rounding values and pop stack values
        ld a1, 8(sp) # Pop t0 to a0
        la a0, displayrne
        call printf
        ld a1, 16(sp) # Now pop t1 onto a1
        la a0, displayrup
        call printf
        ld a1, 24(sp) # Pop t2
        la a0, displayrdn
        call printf
        ld a1, 32(sp) # Pop t3
        la a0, displayrmm
        call printf
        # Restore stack pointer
        addi sp,sp,48
        # exit
        li a7, 93
        ecall
```

Output:

```
$ ./listing8-2
 The result of 123.141592653589797 multiplied by 422.718281828459055 = 52054.202468145471357
 The result of 52054.202468145471357 divided by 123.141592653589797 = 422.718281828459055
 The integer result rounded to nearest (ties to even) is 52054
 The integer result rounded up is 52055
 The integer result rounded down is 52054
 The integer result rounded to nearest (max magnitude) is 52054
```

Listing 8-3 Using dynamic rounding mode

```
# listing 8-3, use of dynamic rounding
.section .data
        pi:         .float 3.141
        formatrup:      .asciz "\n Pi Rounded up result (RUP) is %d\n"
        formatdn:       .asciz "\n Pi Rounded down result (RDN) is %d\n"
        .equ    roundingmask, 0xe0
        .equ    rup,    0x60
        .equ    rdn, 0x40
.section .text
        .global main
        .extern printf
main:
    # Set the rounding mode to round up (0x60)
    frcsr t0                  # Read FCSR into t0
    li    t1, roundingmask    # Mask to clear rounding bits
    not   t1, t1
    and   t0, t0, t1          # Clear bits 7:5
    li    t2, rup
    or    t0, t0, t2
    fscsr t0                  # Write updated FCSR to t0
    # Load Pi into f0
    la    t3, pi
```

```
    flw   f0, 0(t3)
    # Convert to int using dynamic rounding mode (RUP)
    fcvt.w.s a1, f0           # Result goes to a1 (first int argument)
    # Load format string into a0 (first arg for printf)
    la    a0, formatrup
    call  printf
        # Do it again, this time round down
        frcsr t0                  # Read FCSR into t0
        li    t1, roundingmask            # Mask to clear rounding bits
        not   t1, t1
        and   t0, t0, t1          # Clear bits 7:5
        li    t2, rdn            #  0x40 (RDN)
        or    t0, t0, t2
        fscsr t0                  # Write updated FCSR
        # Load Pi into f0
        la    t3, pi
        flw   f0, 0(t3)
        # Convert to int using dynamic rounding mode (RDN)
        fcvt.w.s a1, f0           # Result goes to a1 (first int argument)
        # Load format string into a0 (first arg for printf)
        la    a0, formatdn
        call  printf
        # Return 0 from main
        #li    a0, 0
        #ret
        li a7, 93
ecall
```

Output

```
$ ./listing8-3
 Pi Rounded up result (RUP) is 4
 Pi Rounded down result (RDN) is 3
```

Before looking at the floating-point compare instructions, Listing 8- 4 generates the square root of two numbers. The first number (2) does not have an exact[56] square root, whereas the second number (9) does.

Listing 8-4 Use of sqrt instruction and reading the FCSR register

```
# Listing 8-4.s square root function and reading the fcsr register

# This code could be improved on greatly by showing the actual instruction that flagged the condition

.section .data

message1:      .asciz "\n The square root of the number 9 and 2 when squared is approximately %.14f and %.14f\n"

fcsrerrormsg: .asciz "\n Warning fcsr flags set; the hex value read is %d\n"

accexceptbitsmsg: .asciz "\n 1 = NX (Inexact)\n 2 = UF (Underflow)\n 4 = OF (Overflow)\n 8 = DZ (Divide by zero)\n 10 = NV (Invalid)\n"

square1:      .double 9.0

square2:      .double 2.0

.equ flagmask, 0x1f


.section .text


.global main

.extern printf

main:

       la t0, square1

       la t1, square2

       fld fa0, 0(t0)

       fld fa1, 0(t1)

       fsqrt.d fa2, fa0

       fsqrt.d fa3, fa1

       fmul.d fa4, fa2, fa2

       fmul.d fa5, fa3, fa3

       la a0, message1

       fmv.x.d a1,fa4
```

[56] *The square root of 2 is irrational*

```
      fmv.x.d a2,fa5
      call printf
      li t2, flagmask # Not interested in the rounding bits this time
       frcsr t0        # read fcsr register
       and t0, t0,t2
       beq t0, x0, exit
       la a0, fcsrerrormsg
       mv a1, t0
      call printf
      la a0, accexceptbitsmsg
      call printf
exit:
      li a7, 93
      ecall
```

Output

```
$ ./listing8-4
 The square root of the number 9 and 2 when squared is approximately 9.00000000000000 and
2.00000000000000
 Warning fcsr flags set; the hex value read is 1
 1 = NX (Inexact)
 2 = UF (Underflow)
 4 = OF (Overflow)
 8 = DZ (Divide by zero)
 10 = NV (Invalid)
```

Note that after the instruction `fsqrt.d fa2, fa0` (square root of 9) has been executed the FCSR register looks like:

```
24          fsqrt.d fa2, fa0
25          fsqrt.d fa3, fa1
26          fmul.d fa4, fa2, fa2
27          fmul.d fa5, fa3, fa3
28          la a0, message1
29          fmv.x.d a1,fa4
30          fmv.x.d a2,fa5
31          call printf
32
33          li t2, flagmask # Not interested in the rounding bits
34          frcsr t0        # read fcsr register
35          and t0, t0,t2
```

```
thre Thread 0x3ff7fc3c60 (regs) In: main
g symbols from listing8-4...
o 1
oint 1 at 0x714: file listing8-4.s, line 20.
run
ng program: /home/alan/asm/chapter08/listing8-4
d debugging using libthread_db enabled]
host libthread_db library "/lib/riscv64-linux-gnu/libthread_db.so.1".

oint 1, main () at listing8-4.s:20
n
n
i reg fcsr
        0x0      NV:0 DZ:0 OF:0 UF:0 NX:0 FRM:0 [RNE (round to nearest; ties to even)]
```

No fcsr bits set after the square root of 9 has been calculated

After the instruction `fsqrt.d fa3, fa1` (square root of 2) has completed, GDB shows that the Inexact bit has been set. This will normally indicate that rounding had to be invoked.

```
>    26          fmul.d fa4, fa2, fa2
     27          fmul.d fa5, fa3, fa3
     28          la a0, message1
     29          fmv.x.d a1,fa4
     30          fmv.x.d a2,fa5
     31          call printf
     32
     33          li t2, flagmask # Not interested in the rounding bits
     34          frcsr t0        # read fcsr register
     35          and t0, t0,t2
```

```
multi-thre Thread 0x3ff7fc3c60 (regs) In: main
Reading symbols from listing8-4...
(gdb) b 1
Breakpoint 1 at 0x714: file listing8-4.s, line 20.
(gdb) run
Starting program: /home/alan/asm/chapter08/listing8-4
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/riscv64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at listing8-4.s:20
(gdb) n
(gdb) n
(gdb) i reg fcsr
fcsr           0x0      NV:0 DZ:0 OF:0 UF:0 NX:0 FRM:0 [RNE (round to nearest; ties to even)]
(gdb) n
(gdb) i reg fcsr
fcsr           0x1      NV:0 DZ:0 OF:0 UF:0 NX:1 FRM:0 [RNE (round to nearest; ties to even)]
(gdb)
```

fcsr bit now set after the square root of 2 has been calculated

Care must be taken when making comparisons between floating-point numbers. After a flag has been set in the FCSR register, it is important to note that it must be cleared implicitly by the code. Usually, it is not necessary to check the state of the FCSR register after each floating-point computation has been executed, as the boundaries are usually finite and known in advance.

8.4 Floating-Point comparison instructions

The floating-point comparison instructions are shown below.

Table 8-7 Floating-point comparison instructions

Instruction	Example	Explanation
FEQ.S|D[57]	`feq.d rd, rs1,rs2`	Write the value 1 to the integer register rd, if the double precision number in rs1 is equal to the double precision number in rs2, else write the value 0 to the integer register rd.
FLT.S|D	`flt.s rd, rs1, rs2`	Write the value 1 to the integer register rd, if the single precision number in rs1 is less than the single precision number in rs2, else write the value 0

[57] *Here "|" means or so the instruction could be* `feq.s` *or* `feq.d`

		to the integer register rd.
FLE.S|D	`fle.d rd, rs1, rs2`	Write the value 1 to the integer register rd, if the double precision number in rs1 is less than or equal to the double precision number in rs2, else write the value 0 to the integer register rd.

8.5 Floating-point classification instructions

The classify instructions are used to signify the properties of a floating-point number. There are ten bits available to specify a number's class, only one of these bits set at any given time. Some of these classifications require further explanation, as they were not discussed in chapter one –

- Subnormal A *subnormal* number or a *denormalized* number is a number that is smaller than can be expressed in normal format (1.000…) as described in the IEEE 754 standard. Subnormal numbers are closer to zero than can be expressed in normal format and have less precision.
- Signaling NaN An exception can be raised when NaN is encountered.
- Quiet NaN A quiet NaN does not signal an exception.

Table 8-8 lists the classification bits and their definitions.

Table 8-8 Floating-point classes

Bit	Interpretation when set
0 (1)	Negative infinity -∞
1 (2)	Negative normal
2 (4)	Negative subnormal
3 (8)	Negative zero -0
4 (10)	Positive zero +0
5 (20)	Positive subnormal
6 (40)	Positive normal
7 (80)	Positive infinity +∞
8 (100)	Signaling NaN
9 (200)	Quiet NaN

The next program generates two classes of number – Subnormal and a quiet NaN. The annotated stages to generate the subnormal number are shown in Figure 8- 6.

Listing 8-5 Classification of numbers - subnormal and quiet NaN

```
.section .data
minusone:       .double -1
.section .text
.global _start
_start:
    # Generate a subnormal number by applying division to two normal number
    # For RV64D systems (64-bit registers)
    # First Generate a 64-bit tiny number across t0 and t1
    li t0, 0x00100000   # Upper 32 bits of smallest normal double (2^-1022)
    li t1, 0x00000000      # Lower 32 bits
    # Set up divisor
    li t2, 0x40000000      # Upper 32 bits of 2.0
    li t3, 0x00000000      # Lower 32 bits
    # Consolidate into 64-bit values
    slli t0, t0, 32
    or t0, t0, t1          # t0 = 2^-1022 (smallest normal double)
    slli t2, t2, 32
        or t2, t2, t3          # t2 = 2.0
        # t0 and t2 now have full 64 bit values
        # Store them over to Floating-point registers
        fmv.d.x f0, t0         # f0 = 2^-1022
        fmv.d.x f1, t2         # f1 = 2.0
        # Divide them - produces 2^-1023 (subnormal)
        fdiv.d f2, f0, f1      # f2 = (2^-1022)/2 = 2^-1023
        # Verify the result is subnormal (exp=0, mantissa≠0)
        fmv.x.d t4, f2         # Get bit pattern
        fclass.d t0, f2
        # Expected result: 0x0008000000000000
        # Exponent bits (62:52) = 0
        # Mantissa bits (51:0) = 0x800000000000
```

```
        la t5, minusone
        fld f4, 0(t5)
        fsqrt.d f4, f4  # Square root of -1?
        fclass.d t1, f4
         # Exit
        li a7, 93               # Exit syscall number
        ecall
```

Assemble and link the code with -

```
$  as -g -o listing8-5.o listing8-5.s
$ ld -o listing8-5 listing8-5.o
```

GDB shows the classification of the f2 and f4 registers after computation. Register f2 holds a value smaller than can be represented by normal numbers and is therefore classified as subnormal. Register f4 was used to calculate the square root of minus one which is a complex number and is categorized NaN.

Figure 8-4 GDB showing floating-point number classification

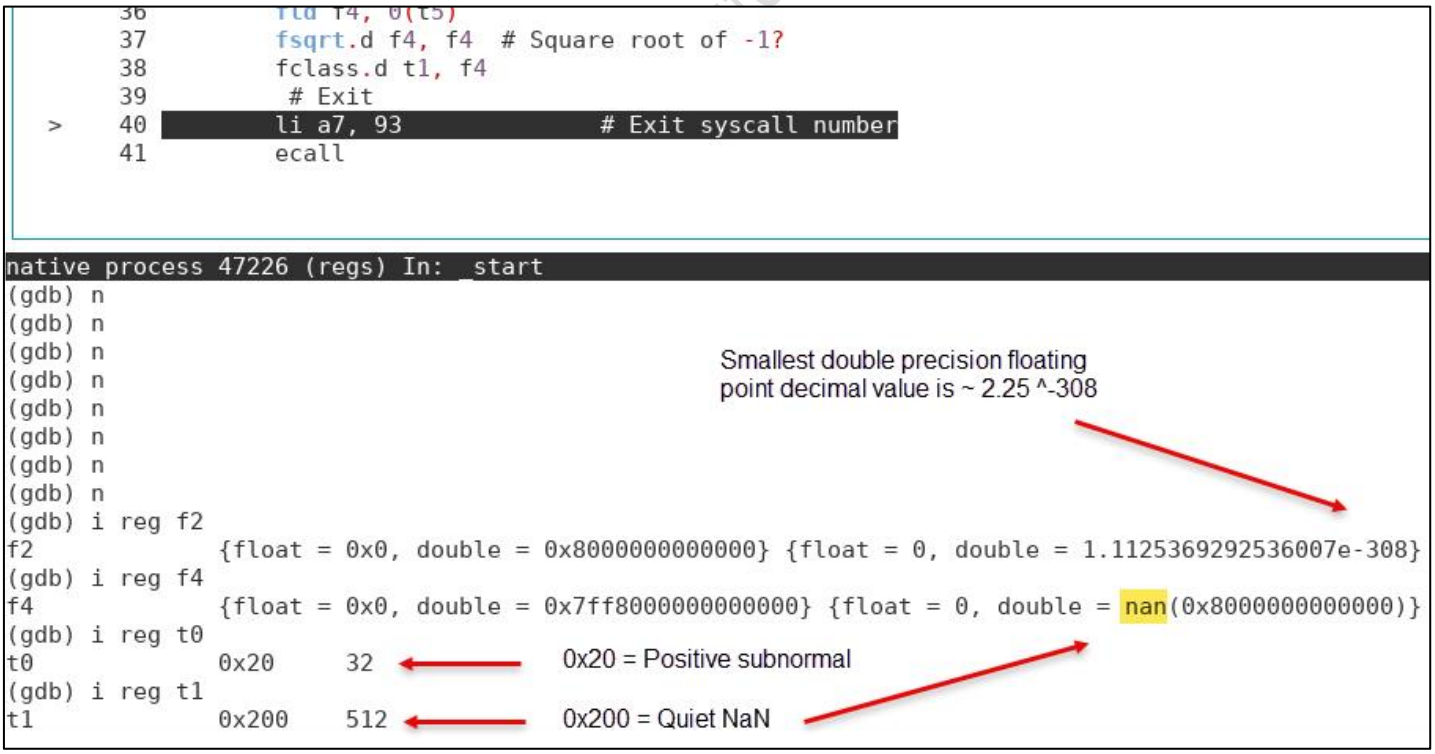
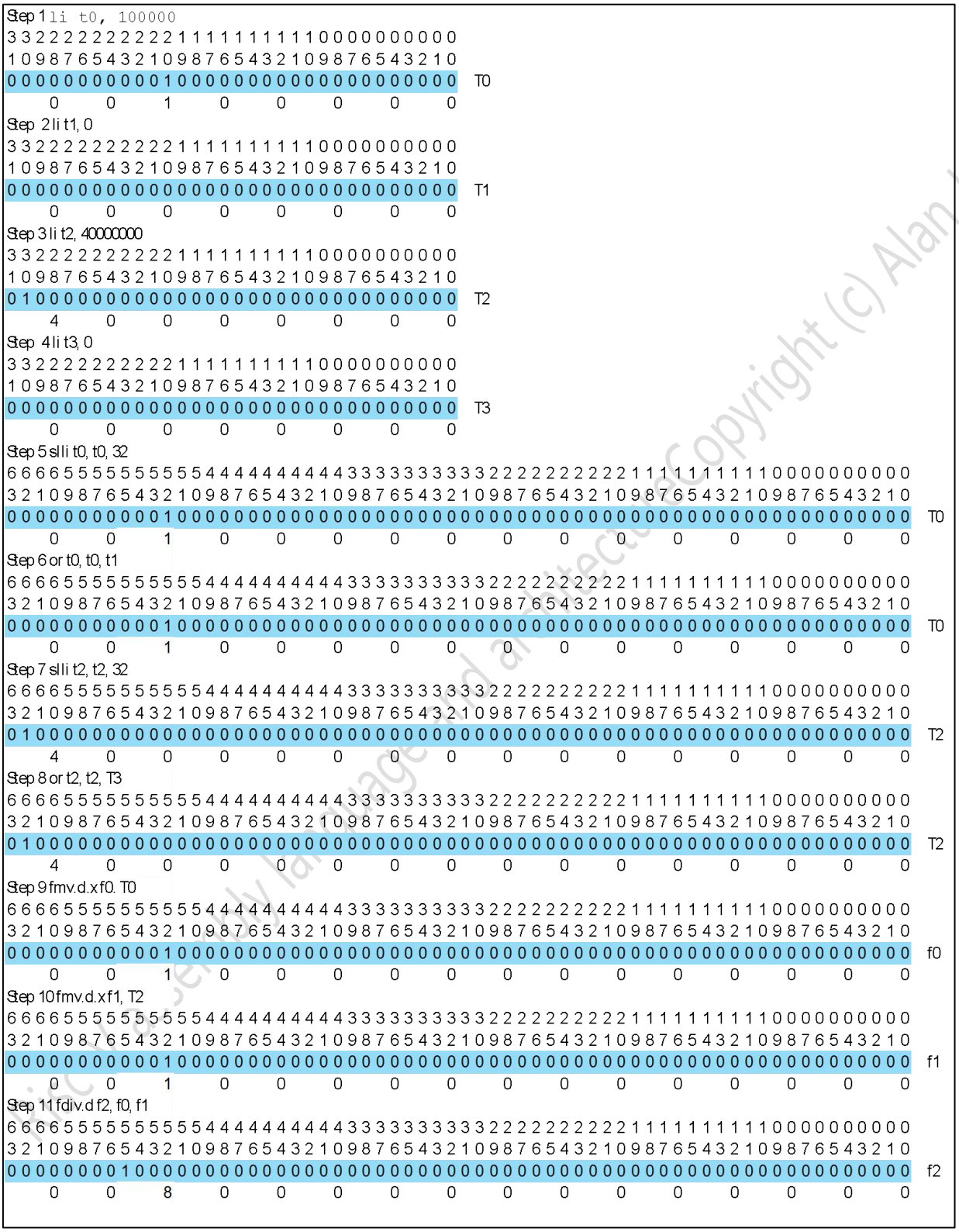


Figure 8-5 Annotated instruction steps to generate a subnormal number



8.6 Exercises for chapter 8

1. Write a program to generate different classes of floating-point numbers and print out the class of the number that was produced.
2. Explain bias as described in IEEE 754

8.7 RISC-V instructions used in chapter 8

8.7.1 Floating-Point Instruction Categories

8.7.1.1 Arithmetic Instructions

Instruction	Operation
fadd.d	Addition
fmul.d	Multiplication
fdiv.d	Division
fsqrt.d	Square root

8.7.1.2 Load / Store Instructions

Instruction	Purpose
fld	Load double
flw	Load single

8.7.1.3 Conversion Instructions

Instruction	Conversion
fcvt.lu.d	Double → unsigned integer
fcvt.w.s	Single float → integer

8.7.1.4 Register Move Instructions

Instruction	Purpose
fmv.x.d	FP → integer register
fmv.d.x	Integer → FP register

8.7.1.5 Comparison Instructions

Instruction	Meaning
feq.d	Equal
flt.s	Less than
fle.d	Less than or equal

8.7.1.6 Classification Instructions

Instruction	Purpose
fclass.d	Determine FP class (NaN, infinity, subnormal, etc.)

8.7.1.7 Floating-Point CSR Instructions

Instruction	Purpose
frcsr	Read FCSR register
fscsr	Write FCSR register

8.7.1.8 Floating-Point Registers Used

Register Type	Examples
Argument FP registers	fa0–fa7
Temporary FP registers	ft0–ft11
General FP registers	f0–f31

8.7.1.9 Rounding Modes Used

Mode	Meaning
rne	Round to nearest even
rup	Round toward +∞
rdn	Round toward −∞
rmm	Round to nearest max magnitude

9 Vector operations

Overview of the chapter

Chapter 9 introduces vector processing in RISC-V using the Vector Extension (V-extension). It explains how to perform SIMD-style operations (Single Instruction, Multiple Data), enabling parallel computation for tasks like matrix math, signal processing, or scientific computing. Vector programming is a complex topic, and many areas are beyond the scope of this document.

This chapter is mainly based on Vector Extension 1.0 which at the time of writing is frozen. The document can be found at the following link at HTTP://github.com/riscvarchive/riscv-v-spec/releases/tag/v1.0

RISC-V "V" Vector Extension

Version 1.0

9.1 Vector system support

The examples shown here were performed on a physical BananaPi BF3 system. The BananaPi SBC has support for vectors as shown by the Linux command below:

```
processor       : 7

hart            : 7

model name      : Spacemit(R) X60

isa                                                                                                 :
rv64imafdcv_zicbom_zicboz_zicntr_zicond_zicsr_zifencei_zihintpause_zihpm_zfh_zfhmin_zca_zcd_zba_zbb
_zbc_zbs_zkt_zve32f_zve32x_zve64d_zve64f_zve64x_zvfh_zvfhmin_zvkt_sscofpmf_sstc_svinval_svnapot_svp
bmt

mmu             : sv39

uarch           : spacemit,x60

mvendorid       : 0x710

marchid         : 0x8000000058000001

mimpid          : 0x1000000049772200
```

The command string to assemble the vector-capable programs used in this chapter is:

```
as  -mno-relax -march=rv64gcv -g -o <filename>.o <filename>.s
```

(indicating that the architecture has *RV64gc*v capability)

Followed by:

```
ld -o <filename> <filename>.o
```

to perform the linking.

If physical hardware is not available, simulators are available[58].

9.2 Vector registers overview

9.2.1 General purpose vector registers

There are 32 vector registers (V0…V31). Vectors can hold scalar values[59] or vector values. The number of *elements*[60] associated with the vector registers is variable and is defined by the total amount of memory available for these vector registers. The number of elements in a vector register is held in the *vector length register*. Arithmetic and logical tasks can be performed, including multiply/divide, floating-point, and shift operations.

Vector registers can be combined into *vector register groups*, allowing a single instruction to operate across multiple vector registers. The *vector length multiplier*, `LMUL`, represents the number of registers that collectively form a vector register group.

LMUL has integer values of 1, 2, 4, and 8.

9.2.2 Vector CSR's

There are seven vector associated CSR registers shown in Table 9- 1[61]

Table 9-1 Vector CSRs

Address	Privilege level	CSR Name	Meaning
0x008	Unprivileged (Read/Write)	vstart	Vector start position
0x009	Unprivileged (Read/Write)	vxsat	Fixed-point saturate flag
0x00A	Unprivileged (Read/Write)	vxrm	Fixed-point rounding mode
0x00F	Unprivileged (Read/Write)	vcsr	Vector control and status register
0xC20	Unprivileged (Read)	vl	Vector length

[58] *See https://github.com/riscvarchive/riscv-v-spec for references to simulation.*

[59] *Integers or floating point.*

[60] *An element is an independent data entity such as the numerator in a division operation.*

[61] *See Vector Extension Programmer's model in volume 1 of the RISC-V instruction set manual for further information*

0xC21	Unprivileged (Read)	vtype	Vector data type
0XC22	Unprivileged (Read)	VLENB	Vector register byte length

9.2.2.1 VSTART register

The *vector start position register (vstart)* is used to specify the index of the first element to be executed by vector instructions. Listing 9- 4 references vector element indices.

9.2.2.2 Vl register

The *vector length register (vl)* contains an unsigned int specifying the number of elements. It is set with the instruction `vset(i)vl(i)` such as `vsetvli t1, t0, e32` where t0 holds the number of elements and e32[62] indicates the elements are 32-bits in size. VL is the number of elements involved in a vector operation.

9.2.2.3 VTYPE register

The *vector data type register* (vtype) indicates the encoding for the *selected element width* (SEW), it occupies bits 5:3 of the vtype register. The SEW bits are defined as shown in Table 9- 2. SEW is the bit size of each individual element withing a vector register.

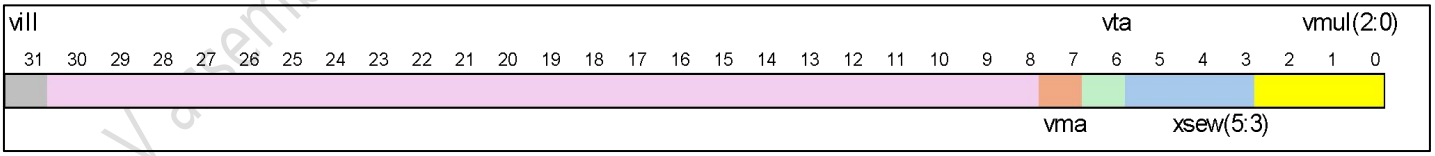
Table 9-2 Vtype SEW bit meaning

VSEW (2:0)	bits		SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64

Bits2:0 represents the vector register group multiplier setting collectively termed LMUL. LMUL has mandatory integer values of 1, 2, 4 and 8. Refer to Table 9- 3 for bit definitions. The register layout is shown in Figure 9- 1.

Fractional values are also supported, such as ½ or ¼[63].

Figure 9-1 Vtype register bit fields



The other bitfield definitions (VILL, VMA and VTA) in the vtype register are discussed later in this chapter.

[62] *Additionally, e8 corresponds to 8 bits, e16 corresponds to 16 bits and e64 corresponds to 64 bits.*

[63] *See the specification for further information and rules.*

9.2.2.4 VLENB register

The vector byte length (VLENB) register has the value VLEN/8, thus representing values in bytes. It is design-implementation dependent, so it could vary by manufacturer. The BananaPi-BF3 (used here) utilizes the SpacemiT K system, which has a fixed VLENB value of 32[64].

- ⇨ VLEN=VLENB*8,
- ⇨ VLEN = 32*8 = 256

VLMAX is defined as LMUL * (VLEN/SEW). It represents the maximum length of the number of elements involved in a single instruction.

Example where VLEN=128 bits

VSEW (2:0)	bits		SEW	Elements per V register
0	0	0	8	16
0	0	1	16	8
0	1	0	32	4
0	1	1	64	2

9.2.2.4.1 Vector suffix

Vector instructions use the `.vv` suffix, such as vadd.vv, to indicate vector operands and the `.vx` suffix to indicate vector and scalar operands. Examples are:

.vv (Vector-Vector): Operates on two vector registers (e.g., `vadd.vv v1, v2, v3).`

.vx (Vector-Scalar): Operates on a vector and a scalar integer register (e.g., `vadd.vx v1, v2, t0).`

.vi (Vector-Immediate): Operates on a vector and a constant (e.g., `vadd.vi v1, v2, 5).`

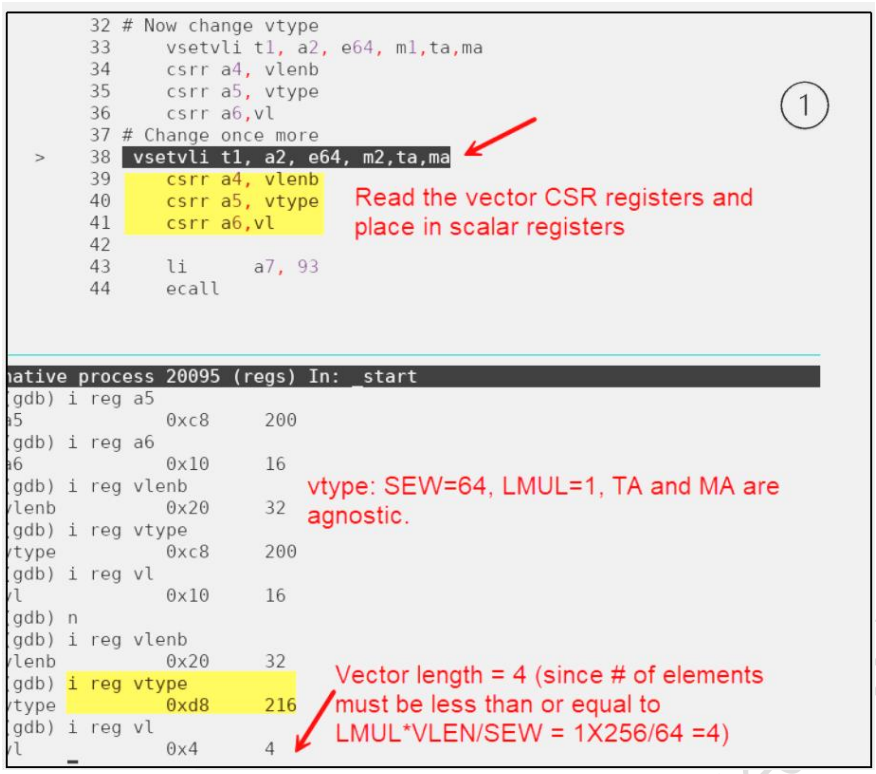
.vf (Vector-Float): Operates on a vector and a floating-point register (e.g., `vfadd.vf v1, v2, fa0).`

Figure 9- 2 shows the vl and vtype states after various `vsetvli` instructions have been executed, here register a2 holds an AVL value of 20[65] with the vl register showing the *actual* number of elements granted.

[64] *The instruction `csrr rd, vlenb` can be used to return the VLENb value in register rd. The instruction `csr` is the control and status read command*

[65] *At the time of writing the default GDB debugger on the BananaPi Bf3 Armbian O/S did not show the vector registers correctly during a debug session. The link https://forum.spacemit.com/t/topic/319?u=alice provided a fix.*

Figure 9-2 Using the CSRR instruction to view Vector CSR values



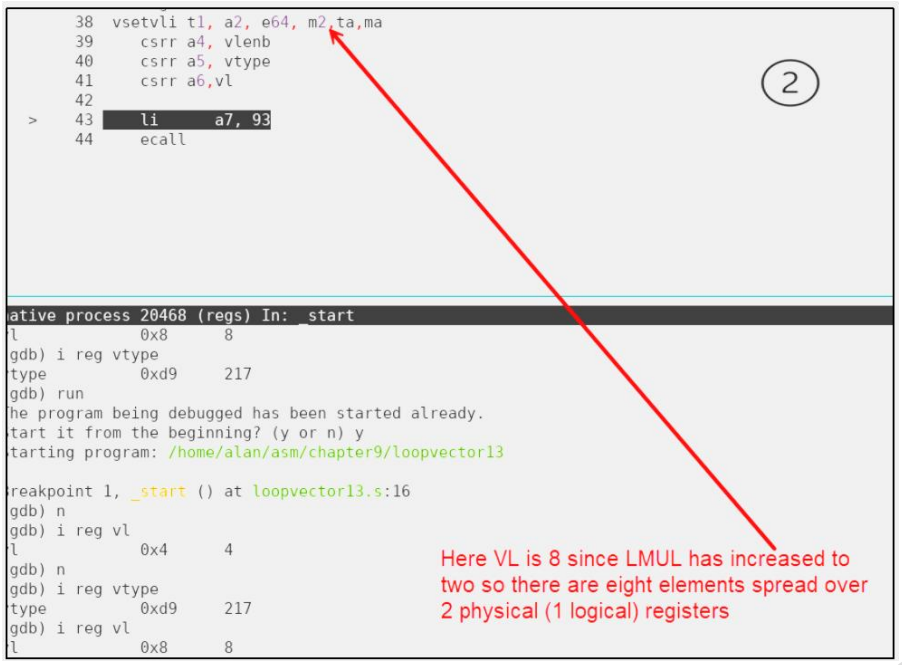
To verify whether the `vsetvli` *parameters are legal (*a non-zero positive number of elements are available given the LMUL/SEW combination*), the legal Vtype value constraint is given as:*

SEW ≤ LMUL*VLEN = 16 ≤ 1*256 is applied. In the first example, SEW ≤ LMUL*VLEN = 16 ≤ 1*256 so it is legal.

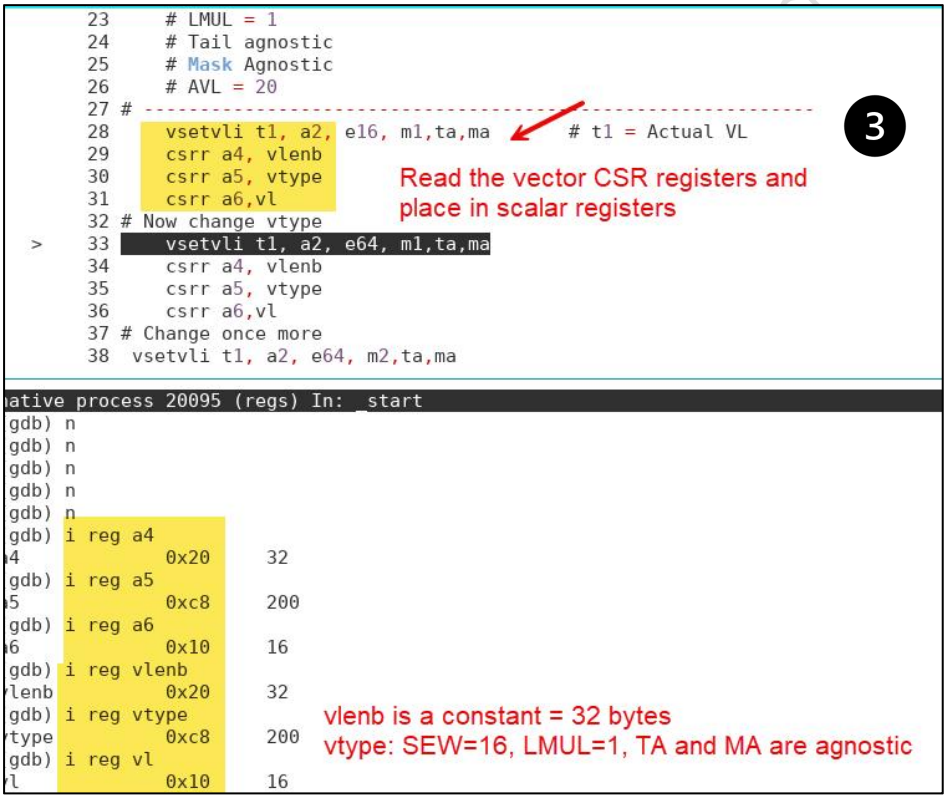
The first configuration setting instruction, `vsetvli t1, a2, e16, m1,ta,ma` requests 20 elements (via register a2) and is granted eight (returned in register t1) as shown in the vl register. This is calculated according to the element capacity constraint: (# of elements * SEW ≤ LMUL*VLEN) or (# of elements ≤ LMUL*VLEN/SEW). Here, # of elements ≤ 1*256/16 so the returned vector length is 16.

The second setting uses a larger element width, and consequently it can only fit 4 elements.

(# of elements ≤ 1*256/64 so the returned vector length is 4.)



The last variant increased LMUL to 2, giving two 32-bit physical vector registers combined into a single 64-bit logical register. The formula is now # of elements ≤ 2*256/16 = 8 and this is reflected in the vl register.



To summarize:

VLENB: The amount of bytes in a vector register

VLEN: Related to VLENB, being the number of bits available in a vector register; this must be a power of two.

ELEN: The maximum element size for a single vector element, must be a power of two.

VL: The number of elements involved in a vector operation

SEW: Defined as the standard element width, (set by `vsetvl` instruction).

LMUL: The vector register grouping value (2,4, or 8) It can also be fractional, with values of 1/8, 1/4,1/2.

Table 9- 3 gives a list of valid values for the BananaPi-BF3 where the Tail and mask attributes are both set to be agnostic.

9.2.2.5 Tail and Mask Attributes

9.2.2.5.1 Tail Attributes

Tail elements are unused elements. So if the VLEN/SEW value is larger than vl (such as 256/16 = 16, and only 10 elements were needed), then the last 6 elements would be unused. These elements past vl are known as tail elements. Here, the used elements would be vn[0]…v[9] and v[10]..v[15] would be the tail elements. The tail elements are not part of the computation.

The remaining tail elements can be set to any value, which is known as *tail agnostic* (ta) or they can remain with their previous value, which is termed *tail undisturbed* (tu). Often, with tail agnostic, the elements might still retain their previous values.

VTA = 0 = Tail undisturbed → Preserve values

VTA = 1 = Tail agnostic →don't care about the values. They might be left undisturbed, zero, garbage, random,...

- Tail elements = lanes beyond VL

9.2.2.5.2 Mask Attributes

Masks are dealt with more comprehensively in 9.5.1, however, mask elements are used with a mask register to decide how to act upon vector register elements. These mask elements are treated similarly to tail attributes in that they can be left undisturbed after an operation or undefined.

VMA = 0 → mask undisturbed (preserved)

VMA = 1 → mask agnostic (don't care)

- Mask elements = lanes disabled by mask

9.2.3 Verify Vector Support

Before starting, we can use the following code to verify that the system does indeed have support for vector operations. This is a programmatic method rather than the Linux method of issuing the command `cat /proc/cpuinfo`.

Table 9-3 Legal Vl value with TA and MA agnostic

Legal Vtype values where VLEN/B=32, with tail and mask both set to agnostic

SEW	LMUL	VILL	Reserved bits 10:8	VMA	VTA	VSEW	VLMUL	VTYPE_bits	VTYPE_hex
8	1/8	0	000	1	1	0	101	11000101	0xC5
8	1/4	0	000	1	1	0	110	11000110	0xC6
8	1/2	0	000	1	1	0	111	11000111	0xC7
8	1	0	000	1	1	0	0	11000000	0xC0
8	2	0	000	1	1	0	1	11000001	0xC1
8	4	0	000	1	1	0	10	11000010	0xC2
8	8	0	000	1	1	0	11	11000011	0xC3
SEW	LMUL	VILL	reserved bits 10:8	VMA	VTA	VSEW	VLMUL	VTYPE_bits	VTYPE_hex
16	1/8	0	000	1	1	1	101	11001101	0xCD
16	1/4	0	000	1	1	1	110	11001110	0xCE
16	1/2	0	000	1	1	1	111	11001111	0xCF
16	1	0	000	1	1	1	0	11001000	0xC8
16	2	0	000	1	1	1	1	11001001	0xC9
16	4	0	000	1	1	1	10	11001010	0xCA
16	8	0	000	1	1	1	11	11001011	0xCB
SEW	LMUL	VILL	reserved bits 10:8	VMA	VTA	VSEW	VLMUL	VTYPE_bits	VTYPE_hex
32	1/8	0	000	1	1	10	101	11010101	0xD5
32	1/4	0	000	1	1	10	110	11010110	0xD6
32	1/2	0	000	1	1	10	111	11010111	0xD7
32	1	0	000	1	1	10	0	11010000	0xD0
32	2	0	000	1	1	10	1	11010001	0xD1
32	4	0	000	1	1	10	10	11010010	0xD2
32	8	0	000	1	1	10	11	11010011	0xD3
SEW	LMUL	VILL	reserved bits 10:8	VMA	VTA	VSEW	VLMUL	VTYPE_bits	VTYPE_hex
64	1/8	0	000	1	1	11	101	11011101	0xDD
64	1/4	0	000	1	1	11	110	11011110	0xDE
64	1/2	0	000	1	1	11	111	11011111	0xDF
64	1	0	000	1	1	11	0	11011000	0xD8
64	2	0	000	1	1	11	1	11011001	0xD9
64	4	0	000	1	1	11	10	11011010	0xDA

Notes:

1. Here, bit 7 and 6 of Vtype are always set since we declared TA and MA to be agnostic
2. Legal Vtype value constraint: SEW/≤ LMUL*VLEN
3. Element capacity constraint: (# of elements * SEW/≤ LMUL*VLEN) or (# of elements ≤ LMUL*VLEN/SEW)
4. Vill is ill legal bit
5. Reserved bits (10:8) must be zero, may be used in future implementations
6. Vtype is a 12bit immediate field

Bit 11	Bit 10:8	Bit 7	Bit 6	Bits 5:3	Bits 2:0
Vill	0	VMA	Vta	VSEW	VLMUL

Listing 9-1 Verifying vector support

```
# Listing 9-1
section .data
warningmsg: .ascii "Warning this program will crash if run on systems without vector support\n"
hasvectormsg: .ascii "This system DOES have vector support\n"
promptmsg: .ascii "Press a key to check\n\n\n\n"
.equ promptmsglen, 22
.equ haslen, 37
.equ warnmsglen, 73
buffer: .space 4
.section .text
global _start
_start:
.option norelax
li a0, 1
la a1,warningmsg
li a2, warnmsglen
li a7, 64
ecall
li a0, 1
la a1,promptmsg
li a2, promptmsglen
li a7,64
ecall
li a0,0
la a1,buffer
li a2,1
li a7, 63
ecall
hasvctspt:
/* Attempt a vector instruction */
.option push
.option arch, +v
```

```
vsetvli zero, zero, e8, m1
.option pop
li a0, 1
la a1, hasvectormsg
li a2, haslen
li a7,64
ecall
li   a0, 1
li a7,93
ecall
```

Generate the executable!

```
$ as -mno-relax -march=rv64gcv -g -o listing9-1.o listing9-1.s
$ ld -o listing9-1 listing9-1.o
```

Output:

```
./listing9-1
Warning this program will crash if run on systems without vector support
Press a key to check


This system DOES have vector support
$
```

On an unsupported system –

```
Warning this program will crash if run on systems without vector support
Press a key to check
Illegal instruction
$
```

9.3 Vector addition/ subtraction example

The next example adds and then subtracts two vector registers; each register contains a total of 8 elements.

- Vector register1 contains values to be added
- Vector register2 contains values to be added
- Vector register3 contains the additive results.
- Vector register4 contains the subtraction results

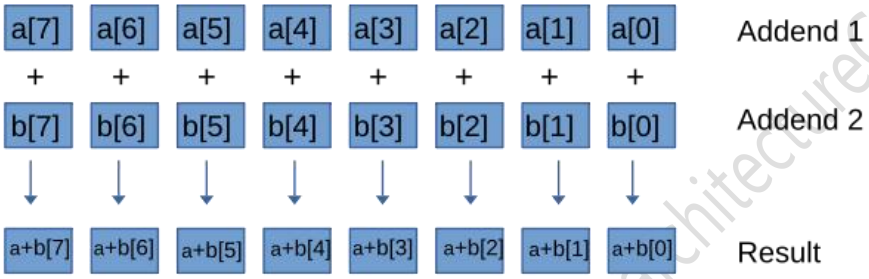
- From earlier `- vsetvli` has the format `vsetvli rd, rs1,imm`

vsetvli command												I-Type instruction																			
3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
imm{11:0]												rs1					1	1	1	rd					1	0	1	0	1	1	1

Immediate Bits			
1	1	VILL	Illegal VTYPE
10:8		Reserved	
7		VMA	Mask Agnostic
6		VTA	Tail Agnostic
5:3		VSEW	SEW in bytes
2:0		VLMUL	LMUL encoding

From the programmer's[66] perspective, the operation takes place in parallel, effectively operating on all the elements of two arrays simultaneously - data1[0], data1[1], .. data1[i] to data2[0], data2[1], …data2[i] and placing the addition of all elements in result[0], result[1],..., result[i]. This is shown in Figure 9- 3.

Figure 9-3 Simultaneous addition of multiple array elements



Listing 9-2 Vector to vector addition/subtraction

```
# Listing9-2.s
# RISC-V Vector Addition and subtraction example
# Adds two vectors with 8 elements each
# Each element is 32 bits in size
.text
.global _start
_start:
   # Configure vector parameters
   li t0, 8               # Set vector length (8 elements)
   vsetvli t1, t0, e32, tu, mu    # Set vector length to 8 (t0), element width to 32 bits (e32)
    # Load vector data (example values)
    la a0, data1     # Load address of first vector
```

[66] *This does not necessarily mean that the instruction is completed during one hardware clock cycle.*

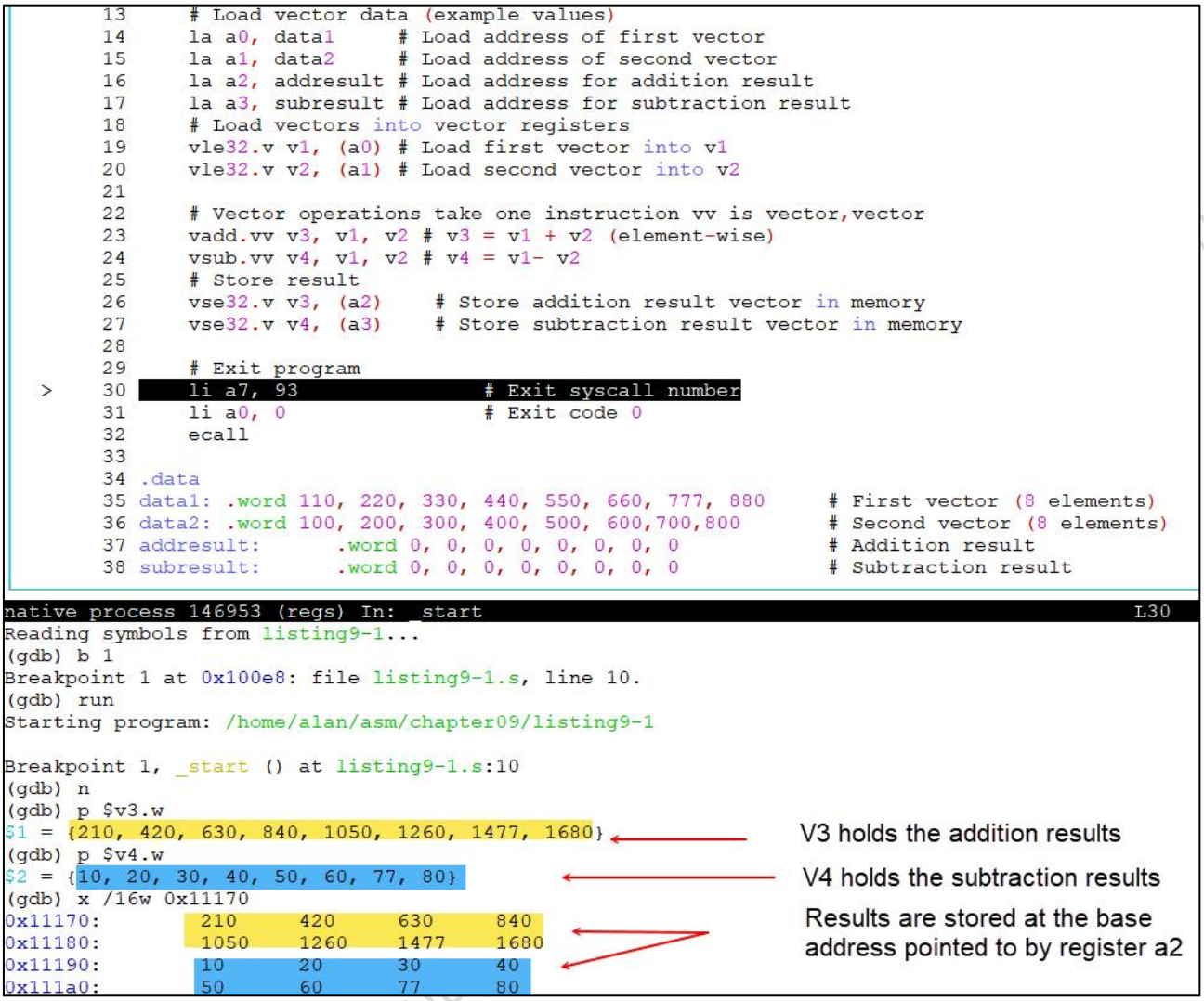
```
    la a1, data2     # Load address of second vector
    la a2, addresult # Load address for addition result
    la a3, subresult # Load address for subtraction result
    # Load vectors into vector registers
    vle32.v v1, (a0) # Load first vector into v1
    vle32.v v2, (a1) # Load second vector into v2
# Vector operations take one instruction vv is vector,vector
    vadd.vv v3, v1, v2 # v3 = v1 + v2 (element-wise)
    vsub.vv v4, v1, v2 # v4 = v1- v2
# Store result
    vse32.v v3, (a2)    # Store addition result vector in memory
    vse32.v v4, (a3)    # Store subtraction result vector in memory
# Exit program
    li a7, 93               # Exit syscall number
    li a0, 0                # Exit code 0
    ecall
.data
data1: .word 110, 220, 330, 440, 550, 660, 777, 880     # First vector (8 elements)
data2: .word 100, 200, 300, 400, 500, 600,700,800       # Second vector (8 elements)
addresult:      .word 0, 0, 0, 0, 0, 0, 0, 0            # Addition result
subresult:      .word 0, 0, 0, 0, 0, 0, 0, 0            # Subtraction result
```

Figure 9- 4 shows the content of the vector registers v3 and v4 which hold the results of the vector addition and vector subtraction operations. The content of the vectors is pushed out to memory via the `vse 32.v` instructions and is shown in `GDB` by examining the location 0x11170 which is pointed to by the integer register a2.

In total, 64 bytes of memory store the two 32-byte vector registers (v3 and V4). The width of the vector registers was set by `e32` in the `vsetvli t1, t0, e32` instruction[67].

[67] *64-bit width is indicated by e64.*

Figure 9-4 GDB showing vector elements.



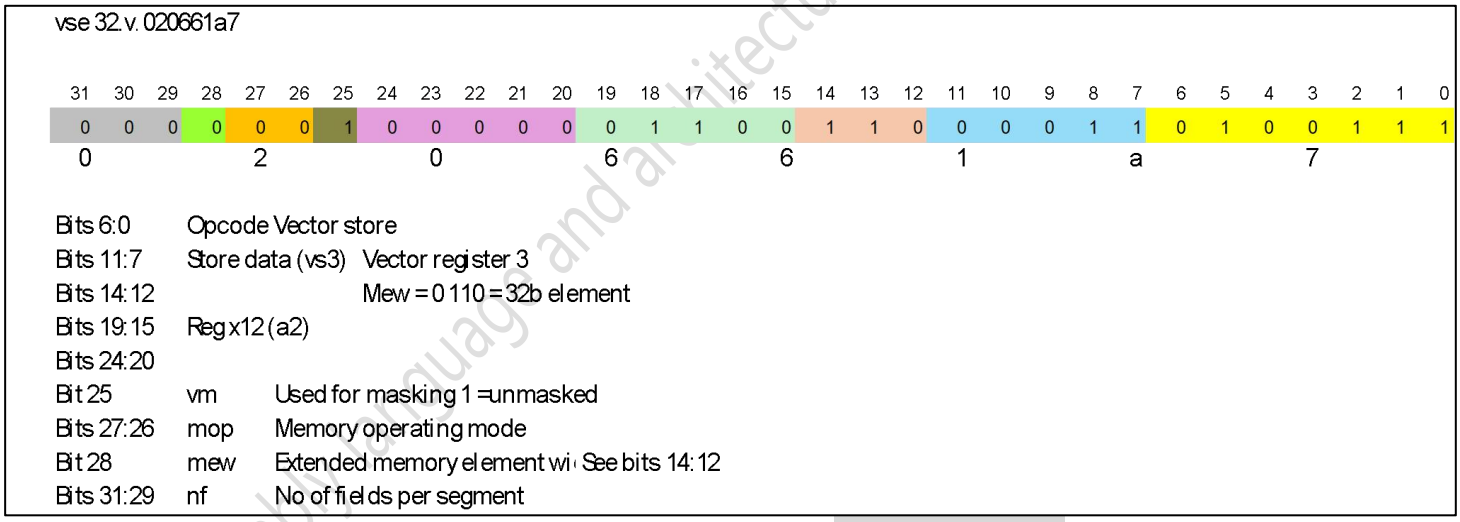
This instruction (`vadd.vv v2, v0, v1`) is an example of a **S**ingle **I**nstruction acting on **M**ultiple pieces of **D**ata (*SIMD*).

Disassembly shows:

```
$ objdump -d -M no-aliases listing9-2

listing9-2:     file format elf64-littleriscv

Disassembly of section .text:

00000000000100e8 <_start>:

   100e8:       42a1                    c.li    t0,8

   100ea:       0102f357                vsetvli t1,t0,e32,m1,tu,mu

   100ee:       00001517                auipc   a0,0x1

   100f2:       04250513                addi    a0,a0,66 # 11130 <__DATA_BEGIN__>

   100f6:       00001597                auipc   a1,0x1
```

```
   100fa:       05a58593                addi    a1,a1,90 # 11150 <data2>
   100fe:       00001617                auipc   a2,0x1
   10102:       07260613                addi    a2,a2,114 # 11170 <addresult>
   10106:       00001697                auipc   a3,0x1
   1010a:       08a68693                addi    a3,a3,138 # 11190 <subresult>
   1010e:       02056087                vle32.v v1,(a0)
   10112:       0205e107                vle32.v v2,(a1)
   10116:       021101d7                vadd.vv v3,v1,v2
   1011a:       0a110257                vsub.vv v4,v1,v2
   1011e:       020661a7                vse32.v v3,(a2)
   10122:       0206e227                vse32.v v4,(a3)
. . .
```

Figure 9-5 Opcode for `vse23.v v3, (a2)`



The opcode breakdown for the vector store instruction V3 → Memory `vse23.v v3, (a2)` is shown in Figure 9-5.

Figure 9- 5 Bit field breakdown for vector store instruction

9.3.1 Adding a vector and a scalar

The next example adds a vector to a scalar. The scalar value is a singular quantity. There are several ways to add a scalar to a vector.

1. Add a scalar from an integer register to a vector register

`vadd.vx` — Vector + Scalar Register

2. Add an immediate value to a vector register

`vadd.vi` — Vector + Immediate

3. Splat + `vadd.vv` (Broadcast)

```
vmv.v.x  v(n), rs1   # broadcast scalar to all elements of a vector
```

```
vadd.vv  vd, vs2, v(n)  # vector + vector
```

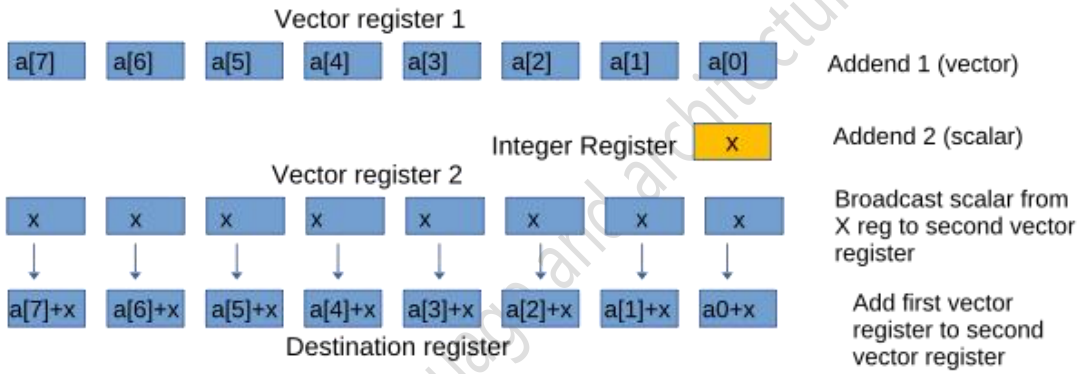
4. Add a scalar FP register with:

`vfadd.vf` (for floats)

Examples are shown in

The first vector register will hold a vector quantity, and the second vector register will hold the scalar. Scalars can be taken from the integer registers or the first element of a vector register. The concept is shown in Figure 9- 6.

Figure 9-6 Adding a scalar to all elements of a vector



The graphic in Figure 9-6 shows a vector register holding an array of eight elements; a scalar quantity is held in an integer register. The content of the integer register is replicated to all elements of a second vector register, and finally both vector registers are added together. The code is shown below.

Listing 9-3 Adding a vector and a vector

```
# Listing 9-3.s

# A scalar from an integer register is broadcast to all elements of another vector register

# Vector to vector addition


.text

.global _start

_start:
```

```
# Configure vector setting
    li t0, 8 # Set vector length to 8 elements
    vsetvli t0, t0, e32, m1, ta, ma # 32-bit elements, vl = 8
# Load Broadcast integer into an integer register (a0)
    li a0, 15
# Generate vector values for v1 rather than obtain them for a .data section
     li t1, 0xffff        # Load integer register t1 with 65535
     vmv.v.x v1, t1       # Set all elements to 0xffff (the value in t1)
     li t1, 11
# vid.v is the vector element index instruction, each element's index is written from
# 0 to the vector length -1, since vl = 8, 0-7 are written to the destination register (v0)
      vid.v v0             # indices (0,1,2...) into v0
      vmul.vx v0, v0, t1   # Multiply indices by t1 value, v0=0,11,22,...
      vadd.vv v1, v1, v0   # v1 = [65535, 65546, 65557, 65568, 65579, 65590, 65601, 65612]
# Move scalar in x register a0 to each vector element (broadcast/splat)
     vmv.v.x v3, a0       # Broadcast scalar in a0 to all elements of v3,v3=15,15,...
# Vector-vector addition (v2 = v1 + v3)
     vadd.vv v2, v1, v3   # v2[i] = 65550, 65561, 65572,...
        li a7, 93            #      Invoke syscall
        ecall
```


The program explanation is as shown.

The vector settings are configured for eight elements with a width of 32 bits. The code loads a scalar value 0xffff (65535) into the integer register t1. The instruction `vmv.v.x v1, t1` moves the value held in the x register t1 to all elements of the vector register v1.

a. Each element in v1 is now {65535, 65535, 65535, 65535, 65535, 65535, 65535, 65535}.

The vid.v v0 instruction writes each element's *index* to the destination; the indices range from 0 to (vector length – 1).

b. Since v0 is the destination and the vector length has been set to 8, v0 is now {0,1,2,3,4,5,6,7}.

The indices are multiplied by the value of 11 (register t1) with the instruction vmul.vx v0, v0, t1, giving a result of {0, 11,22,33,44,55,66,77} in vector v0.

Vector v0 and vector v1 are added together, giving v1 the result {65535, 65546, 65557, 65568, 65579, 65590, 65601, 65612}.

The scalar value 15 is replicated (broadcasted) to all elements of v3.

Giving v3 the value {15,15,15,15,15,15,15,15}. Next, a vector addition (,vv) is performed on v1 and v3, with the result going into v2. V2 has the values 65550, 65561, 65572, …

This next program shows the previous and other types of vector addition:

Listing 9-4 Different methods for vector addition

```
 .section .data
.align 4
scalararray:      .word 1,2,3,4,5,6,7,8
.align 4
floatparray:     .float 1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0
    .section .text
    .globl _start
_start:
# Setup vector length: 8 elements, 32-bit each
    li      a0, 8
    vsetvli t0, a0, e32, m1,ta,ma
# Load integer vector
    la      a1, scalararray
    vle32.v v1, (a1)
#v1 now holds [1] [2] [3] [4] [5] [6] [7] [8]
    li t1, 5        # scalar = 5

# Addition Method 1  Vector-Scalar Add
    vadd.vx v2, v1, t1     # v2 = v1 + 5, note .vx suffix
# v2 = [6] [7] [8] [9] [10] [11] [12] [13], scalar added to all elements

# Addition Method 2. Vector-Immediate Add signed 5-bit (-16..+15)
    vadd.vi v3, v1, 10      # v3 = v1 + 10, note .vi suffix
# V3 = [11] [12] [13] [14] [15] [16] [17] [18]

# Method 3.  Vector-Vector Add
    vadd.vv v4, v2, v3  # v3 = v2 + v3, note .vv suffix
# v4 = [17] [19] [21] [23] [25] [27] [29] [31]

# Method 4. Floating-Point Scalar Add
    la a2, floatparray
```

```
    vle32.v v5, (a2)
    li t2, 0x40A00000     # 5.0f (IEEE-754 form)
    fmv.w.x ft0, t2 # Move the bits from t2 into ft0
    vfadd.vf v6, v5, ft0       # note .vf suffix v6 = v5 + 5.0 = [6],[7],…


    # Exit
    li      a7, 93
    ecall
```

9.4 Vector Permutations

There are a number of vector permutation-related instructions:

- Slide

Vector slide instructions in RVV are similar to left/right, but done lane-wise -

`vslideup.vx / vslideup.vi` — shift elements toward higher indices

It shifts existing elements upward. The lower lanes become zero .

An example (offset=4):

src: [10 11 12 13 14 15 16 17]

dst: [0 0 0 0 10 11 12 13]

`vslidedown.vx / vslidedown.vi` — shift elements toward lower indices

Its function is to drop elements from the bottom, shift down, and fill the top lanes with zeros.

An example (offset=2):

src: [10 11 12 13 14 15]

dst: [12 13 14 15 0 0]

```
      li a7, 93
      ecall
```

9.4.1 Moving elements with vslide

The next example adds individual elements from two vector registers. This is accomplished by extracting the individual elements from the vector registers, placing them in scalar registers, and then performing a scalar addition. This is accomplished by the `vslidedown` instruction. For completeness, the `vslideup` instruction is included.

The elements are actually moved by the `vslide instructions,` which "slides" elements by a number of positions; elements that have been slid out are replaced by zeros. This is similar to shift/rotate operations. A further example is shown.

Listing 9-5 Use of vector vslide instructions

```
# Listing9-5.s
.section .data
.align 4
vector1: .word 10, 20, 30, 40, 50, 60, 70, 80
vector2: .word 1, 2, 3, 4, 5, 6, 7, 8
.text
.global _start
_start:
       li t1, 8 # number of elements
# Load vector from memory
       la a0, vector1
       la a1, vector2
       vsetvli t0, t1, e32, m1, ta, ma  # 8 elements, 32-bit each
       vle32.v v1, (a0)     # v1 = [10,20,30,40,50,60,70,80]
       vle32.v v2, (a1)     # v2 = [1,2,3,4,5,6,7,8]
# Move up 4 places
       vslideup.vi v3, v1, 4
# Move down
       vslidedown.vi v4, v2, 2
# V3 now[0, 0, 0, 0, 10, 20, 30, 40]
# V4 now[3, 4, 5, 6, 7, 8, 0, 0]
       vmv.x.s t2, v3          # t2 now holds 0, (first element of v3[0])
       vmv.x.s t3, v4          # t3 now holds 3 v4[0]
# Exit
        li a7, 93
        ecall
```

Consider the instruction `vslideup.vi v3, v1, 4` in the listing. Initially vector register 1 contains the eight elements [10, 20, 30, 40, 50, 60, 70 80] and they will be “slid” 4 places upwards (to the right) into vector register V3. As the elements are moved leftwards, they are replaced from the right by zeros, with the result being placed in vector register v3.

`Vslidedown` moves the elements leftwards, padding from the right.

```
(gdb) p $v1.w
$2 = {10, 20, 30, 40, 50, 60, 70, 80}
(gdb) p $v2.w
$3 = {1, 2, 3, 4, 5, 6, 7, 8}
(gdb) p $v3.w
$4 = {0, 0, 0, 0, 10, 20, 30, 40}
(gdb) p $v4.w
$5 = {3, 4, 5, 6, 7, 8, 0, 0}
(gdb) i reg t2
t2             0x0        0
(gdb) i reg t3
t3             0x3        3
```

9.4.2 Vrgather

The vslide instructions shift elements by a fixed offset, such as `vslidedown.vi v4, v2, 2(offset)` whereas the `vrgather` instructions select elements by an index, placing them in a destination. Variants are shown in Table 9-4.

Table 9-4 vrgather variants

Format	Type	Broadcast	Comments	Syntax
VRGATHER.VI	Immediate	Yes	Dest, source, index (immediate)	`vrgather.vi v3, v1, 5`
VRGATHER.VX	Scalar	Yes	Dest, source, index (register)	`vrgather.vx v4, v2, t3`
VRGATHER.VV	Vector	No	Dest, source, index list (vector)	`vrgather.vv v5, v1, v0`

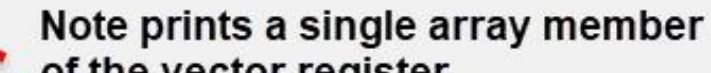
An example of this instruction is shown in Listing 9-6

Listing 9-6 vrgather instruction

```
# Listing9-6 illustrates the vrgather instruction
    .section .data
    .align 4


    vec0: .word 10, 20, 30, 40, 50, 60, 70, 80
    vec1: .word 100, 200, 300, 400, 500, 600, 700, 800
    .text
    .global _start
    _start:
        li t1, 8 # number of elements
        li t3, 3
```

```
    # Load vector from memory
        la a0, vec0
        la a1, vec1
        vsetvli t0, t1, e32, m1, ta, ma  # 8 elements, 32-bit each
        vid.v v0    # write index values to v0
        vle32.v v1, (a0)     # v1 = [10,20,30,40,50,60,70,80]
        vle32.v v2, (a1)     # v2 = [100,200,300,400,500,600,700,800]
    # vrgather allows for direct gather of a single lane
    # Here lane 5 and lane 3 of vector registers v1 and v2 are sent to vector lane of the vector
registers 3 and 4
        vrgather.vi v3, v1, 5 #Immediate version
        vrgather.vx v4, v2, t3 # Register version
        vrgather.vv v5, v2, v0 # v0 holds the index, v1 is the source, v5 is vd
    # V3 looks like [60, 60, 60, 60, 60, 60, 60, 60]
    # V4 looks like [400, 400, 400, 400, 400, 400, 400, 400]
    # V5 looks like [100, 200, 300, 400, 500, 600, 700, 800] # Note indices in V0
        vmv.x.s t2, v3          # t2 now holds 60, first element of v3[0]
        vmv.x.s t3, v4          # t3 now holds 400 v4[0]


        # Exit
        li a7, 93
        ecall
```

```
(gdb) p $v0.w
$12 = {0, 1, 2, 3, 4, 5, 6, 7}
(gdb) p $v1.w
$13 = {10, 20, 30, 40, 50, 60, 70, 80}
(gdb) p $v2.w
$14 = {100, 200, 300, 400, 500, 600, 700, 800}
(gdb) p $v3.w
$15 = {60, 60, 60, 60, 60, 60, 60, 60}
(gdb) p $v4.w
$16 = {400, 400, 400, 400, 400, 400, 400, 400}
(gdb) p $v5.w
$17 = {100, 200, 300, 400, 500, 600, 700, 800}
(gdb) p ((int[8])$v1)[5]
$18 = 60
(gdb) p ((int[8])$v2)[3]
$19 = 400
(gdb) i reg t2
t2             0x3c     60
(gdb) i reg t3
t3             0x190    400
```



The next example shows how to reverse the order of 16 elements in a register. The indices held in v0 are reversed. This means that a copy from vector source to vector destination will be copied in the order held in v0, thus reversing the source register's elements.

In this example, v1 is the source register and v2 is the destination.

Listing 9-7

```
#Listing 9-7 # Using vrgather instruction to reverse a register's elements

# Listing9-6 illustrates the vrgather instruction

.section .data

.align 4

vec0: .short 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0

vec1: .short 10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85

.text

.global _start

_start:

    li t1, 16 # number of elements

# Load vector from memory

    la a0, vec0

    la a1, vec1
```

```
    vsetvli t0, t1, e16, m1, ta, ma  # 16 elements, 16-bits each

    vle16.v v0, (a0)

    vle16.v v1, (a1)     # v1 = [10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85]

# vrgather allows for direct gather of a single lane

# The indices of v0 are in reverse order causing the source vector register's element to be copied
in reverse

    vrgather.vv v2, v1, v0 # v0 holds the index, v1 is the source, v2 is the destination

    #vmv.x.s t2, v3          # t2 now holds 0, first element of v3[0]

    #vmv.x.s t3, v4          # t3 now holds 3 v4[0]


    # Exit

    li a7, 93

    ecall
```

```
(gdb) p $v0.s
$1 = {15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
(gdb) p $v1.s
$2 = {10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85}
(gdb) p $v2.s
$3 = {85, 80, 75, 70, 65, 60, 55, 50, 45, 40, 35, 30, 25, 20, 15, 10}
(gdb)
```

9.5 Grouping vector registers

When working with certain datasets, it is often not necessary to have 32 vector registers. This might be the case when dealing with comparisons of data held in just two vector registers. Rather than comparing eight elements at a time (assuming that 8 is the maximum number of elements having the required data size that can be accommodated by a single vector register, the data size) it could be more convenient to process sixteen elements (or more) with each instruction.

By grouping vector registers, the model presented to the programmer might be 16 registers, each having 16 elements, or 8 registers with 32 elements, etc.

Figure 9-7 It shows the concept where two 8-element vector registers are combined into one 16-element vector register. Grouping is accomplished by the instruction `vsetivli t0, 16. E32, m2,` here `m2` signifies that the number of groups is 16 (32/2), a value of 4 represents 8 groups, and a value of 8 would represent 4 groups. This is shown in Table 9- 3.

Table 9- 5 LMUL and grouping correspondence

VLMUL (2:0)	LMUL	# of groups
000	1	32
001	2	16

010	4	8
011	8	4

The grouped register is addressed as a single operand using the first grouped register, so an instruction such as `vle32.v v0, (t0)` would load the data pointed to by t0 into registers v0 and v1.

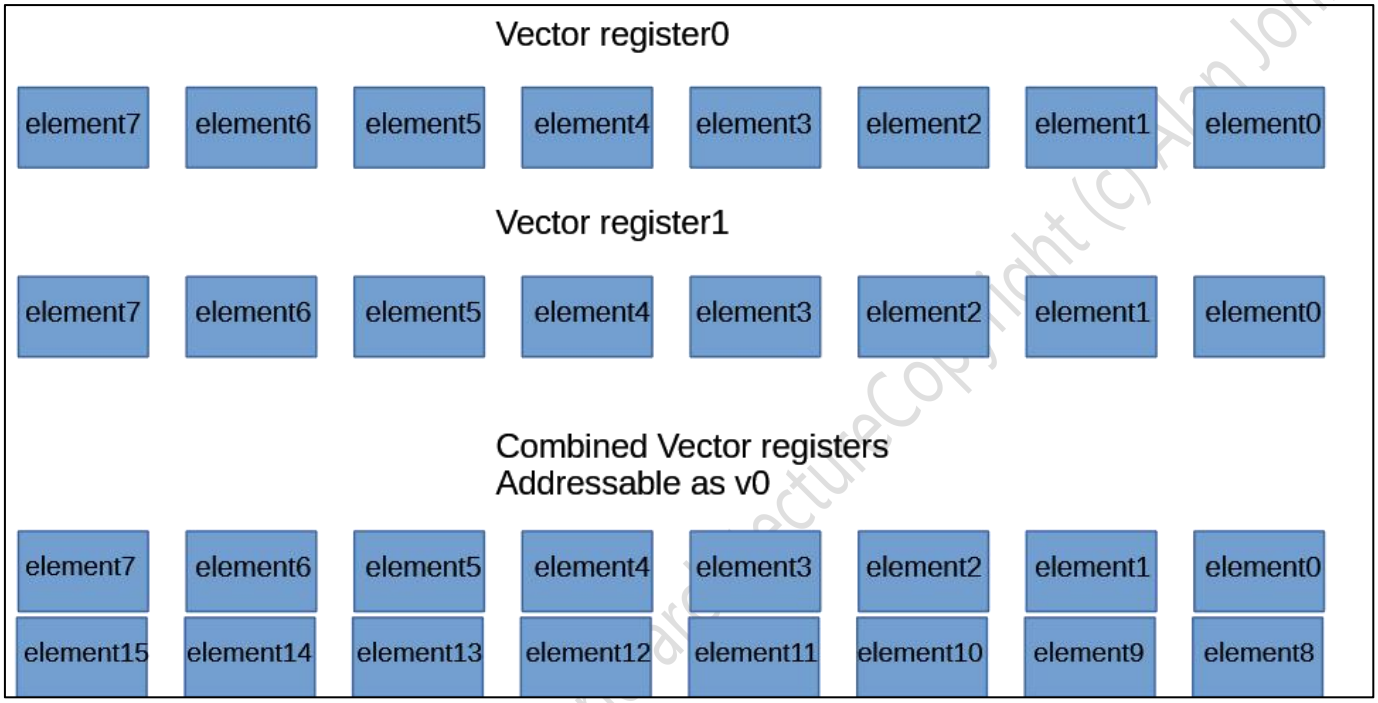


Figure 9-7 Grouping vector registers

The next program shows how to combine vector registers into groups of two.

Listing 9-8 Grouping vector registers

```
# Listing 9-8.s
# Groups two vectors together as one
# V0 and V1 form one group addressed by v0
# V2 and V3 form the second group addressed by V2
# . . .
.section .data
# First vector's contents
dataset1: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
# Second vector's contents
dataset2: .word 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
.section .text
```

```
.global _start

_start:

# Configure for LMUL=2 with m2 (group 2 registers together)

        vsetivli t0, 16, e32, m2  # 16 elements (2x8), 32-bit, LMUL=2

# Recall LMUL represents the grouping factor

# Load first dataset into v0 (first register in group)

        la t1, dataset1                 #Point to dataset1

        vle32.v v0, (t1)        # Address group by first vector in the group (v0)

# Load second dataset into the next group)

        la t1, dataset2         #Point to dataset2

# Address group by first vector in the group (v2)

        vle32.v v2, (t1)

# v0-v1: First vector

# v2-v3: Second vector

#Process each group as single 16-element vector:

# Example :

# Add 2 to all 16 elements of the first grouped vector

# Add 3 to all 16 elements of the second grouped vector

# Use vector integer add instruction

    vadd.vi v0, v0, 2      # Add 2 to first vector

    vadd.vi v2, v2, 3       #Add 3 to second vector

# Exit

    li a7, 93

    ecall
```

The instruction `vsetivli t0, 16, e32, m2` includes m2 to set the grouping, previous instances of this instruction did not include an m value which left the default group at 1 → 1 register corresponding to 1 group. The e32 designation is the element size → 32 bits and the preceding number → 16 is the number of elements.

Figure 9-8 shows the vector registers before and after the data has been loaded.

Note that a single instruction loads 16 elements; without grouping two instructions, would be required – the first instruction to load vector 0 and the second to load vector 1.

Similarly, each `vadd` instruction operates on all of 16 elements of each register pair with one instruction.

Figure 9-8 Loading two vector registers with one instruction

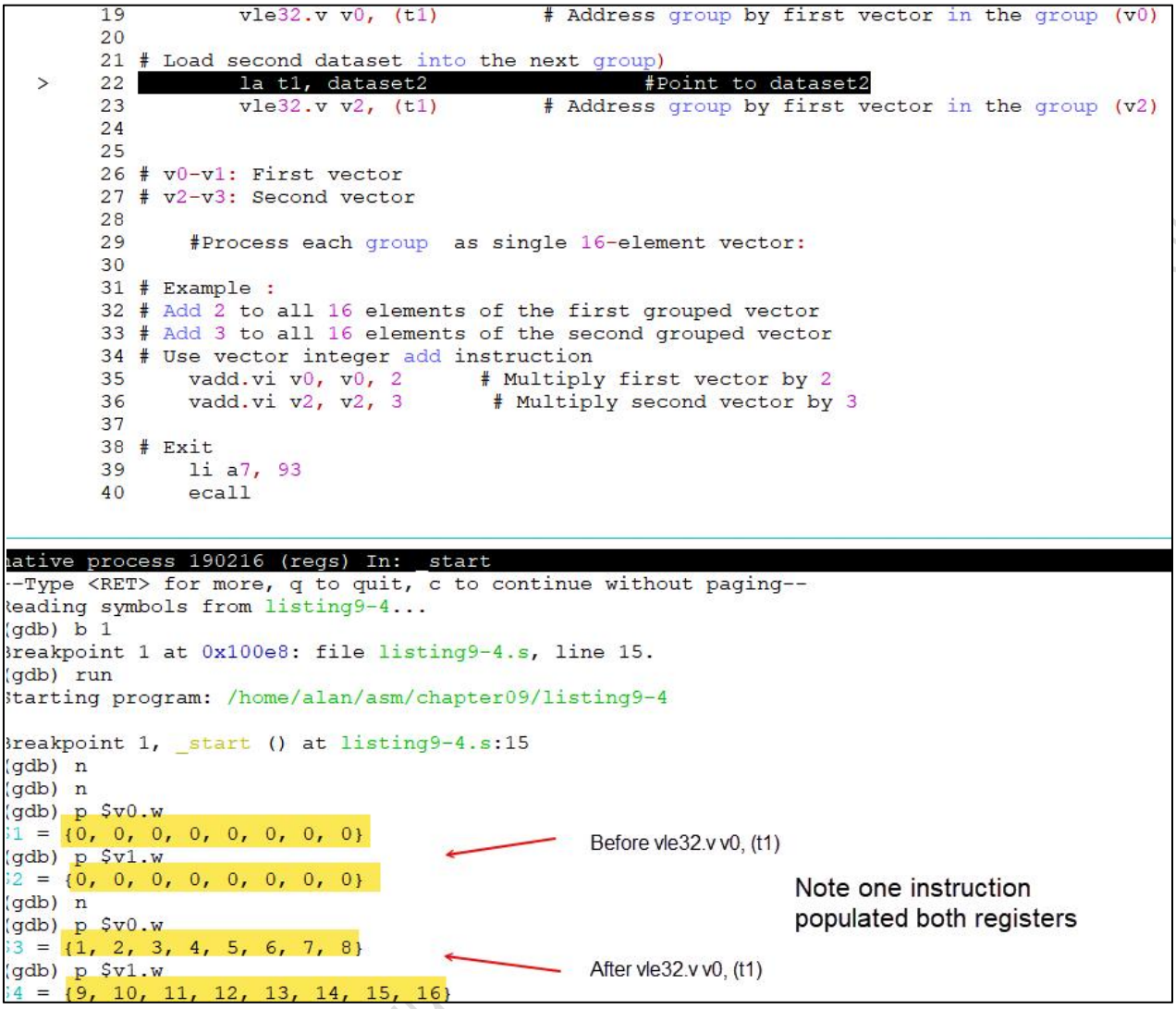
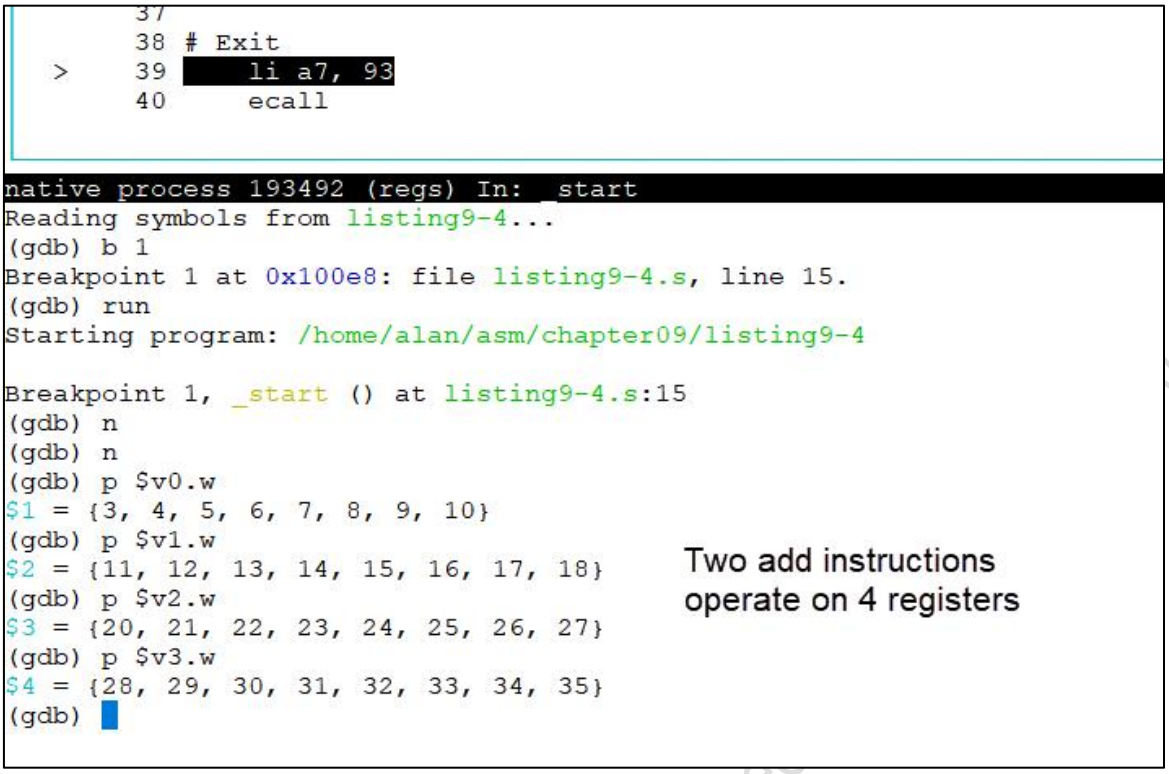


Figure 9-9 Operating on two vector registers with a single add instruction



After the `vsetivli t0, 16, e32, m2` has been executed, the CSR registers show the values listed in Figure 9-10. The value 0x11 in the vtype register gives the SEW bits (5:3) as 010 and the vmul bits as 001

Figure 9-10 CSR registers after execution of the vsetivli t0, 16, e32, m2 instruction

vl	0x10	16
vtype	0x11	17
vlenb	0x20	32

9.5.1 Masking and merging

RISC-V can *merge* elements from two vectors based on certain conditions. A *mask* can be used so that a value can be taken from the first source register if a Boolean is true or from the second source register if the Boolean is false with the result going to a destination register. For example, a mask consisting of 1,1,0,1,0,1 would take the first two values from rs1, the next value from rs2, the fourth from rs1, the fifth from rs2 and finally the sixth from rs1.

The example below shows V0 as the mask register, v1 and v2 as the source registers, and v3 as the destination register.

V0 holds 0xAA = ob10101010

V1 holds 1,3,5,7,9,11,13,15

V2 holds 2,4,6,8,10,12,14,16

After the merge based on V0's mask -

V3 holds 1,4,5,8,9,12,13,16.

This is shown graphically in Figure 9-11.

Figure 9-11 showing vmerge with masking

Rules	mask=0 -> take element from v1
	mask=1 -> take element from v2
Note:	Bit n is associated with element n

V0 Mask bits	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit0
	1	0	1	0	1	0	1	0

	Element7	Element6	Element5	Element4	Element3	Element2	Element1	Element0
V1	15	13	11	9	7	5	3	1
V2	16	14	12	10	8	6	4	2
V3	16	13	12	9	8	5	4	1

The listing shows how to use vmerge.

Listing 9-9 Use of vmerge instruction

```
# Listing 9-9.s
# Use of mask and vector vmerge instruction
.section .data
        oddnumbers:   .word 1, 3, 5, 7, 9, 11, 13, 15
        evennumbers:   .word 2, 4, 6, 8, 10, 12, 14, 16
        result: .space 16
.section .text
.global _start
_start:
# Set VL (vector length) to 8 elements
        li      t0, 8
        vsetvli t0, t0, e32, m1, ta,may


# Load oddnumbers into v1
        la      a1, oddnumbers
        vle32.v v1, (a1)
# Load evennumbers into v2
        la      a2, evennumbers
```

```
        vle32.v v2, (a2)
# Set up a mask register: this will select alternate elements odd and even
# Use v0 with binary pattern: to hold mask bits
        li      t1, 0b1010101010101010
        vmv.v.x v0, t1
        vmerge.vvm v3, v1, v2, v0
# Store the result
        la      a3, result
        vse32.v v3, (a3)
# Exit
        li      a7, 93
        ecall
```

```
(gdb) p /x $v0.w
$6 = {0xaaaa, 0xaaaa, 0xaaaa, 0xaaaa, 0xaaaa, 0xaaaa, 0xaaaa, 0xaaaa}
(gdb) p /x $v1.w
$7 = {0x1, 0x3, 0x5, 0x7, 0x9, 0xb, 0xd, 0xf}
(gdb) p /x $v2.w
$8 = {0x2, 0x4, 0x6, 0x8, 0xa, 0xc, 0xe, 0x10}
(gdb) n
(gdb) p /x $v3.w
$9 = {0x1, 0x4, 0x5, 0x8, 0x9, 0xc, 0xd, 0x10}
```

GDB shows the content of V3 after the merge has been completed.

```
listing9-5.s
       20
       21 # Set up a mask register: thsi will select alternate elements odd and even
       22 # Use v0 with binary pattern:  to hold mask bits
       23         li      t1, 0b1010101010101010
       24         vmv.v.x v0, t1
       25
       26 # vmerge.vvm result into v3 = merge v1 and v2 based on mask v0
       27         vmerge.vvm v3, v1, v2, v0
       28
       29 # Store the result
       30         la      a3, result
       31         vse32.v v3, (a3)
       32
       33 # Exit
  >    34         li      a7, 93
       35         ecall
       36
```

```
native process 4218 (regs) In:  _start
Reading symbols from listing9-5...
(gdb) b 1
Breakpoint 1 at 0x100e8: file listing9-5.s, line 10.
(gdb) run
Starting program: /home/alan/asm/chapter09/listing9-5

Breakpoint 1, _start () at listing9-5.s:10
(gdb) n
(gdb) p $v1.w
$1 = {1, 3, 5, 7, 9, 11, 13, 15}
(gdb) p $v2.w
$2 = {2, 4, 6, 8, 10, 12, 14, 16}
(gdb) p $v3.w
$3 = {1, 4, 5, 8, 9, 12, 13, 16}
(gdb) x /16w 0x11168
0x11168:        1       4       5       8
0x11178:        9       12      13      16
```

Memory

The next example uses a bitmask to add certain elements of vector registers together. In this example, the bitmask is loaded into a vector and then the instruction, vector mask if not equal, is illustrated. This is different from the earlier listing where an integer register was used.

Listing 9-10 Adding selected elements in vector registers with a bitmask

```
# Listing 9-10

.section .data

# Alternating mask pattern (byte per element)

.align 4

bitmask:
```

```
    .byte 1,0,1,0,1,0,1,0
# Input vectors (8 elements)
.
align 4
vectorA:
    .word 10,20,30,40,50,60,70,80
.align 4
vectorB:
    .word 1,2,3,4,5,6,7,8
# Output buffer
.align 4
result:
    .space 32              # 8 * 4 bytes
    .section .text
    .global _start


_start:
    li a0, 8 # Number of elements
    la a1, bitmask
    la a2, vectorA
    la a3, vectorB
    la a4, result
#  Load vectors (e8)
    vsetvli t0, a0, e8, m1, ta, ma
    vle8.v v0, (a1)# Loads vl elements, each element = 1 byte, v0 = [1,0,1,0,...]
    vmsne.vi v0,v0,0 # Converts to a mask, writes a 1 where elements are not equal, otherwise a
zero
# Use this instruction to invert the bits - (vmnot.m v0,v0)
# Load vectors (e32)
    vsetvli t0, a0, e32, m1, ta, ma
    vle32.v v1, (a2)           # v1 = vectorA
    vle32.v v2, (a3)           # v2 = vectorB
# Add using the mask
    vadd.vv v3, v1, v2, v0.t   # only odd elements updated
```

```
# Store result
    vse32.v v3, (a4)
    li a7, 93                   # syscall: exit
    ecall
```

9.5.2 Matrix Inversion Program

This program will generate the inverse of a 2x2 square matrix. Prior to examining the code, it may be helpful to discuss the method used below.

9.5.2.1 Steps to generate the inverse of a square matrix:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Matrix A =

$$\begin{pmatrix} 4,3 \\ 11,9 \end{pmatrix}$$

- The formula is A^{-1} = 1/detA X adjugate of A.
- The determinant is (4 x 9)-(3 X 11) = 3
- The adjugate swaps a and d and negates b and c
- The adjugate is $\begin{pmatrix} 9.-3 \\ -11,4 \end{pmatrix}$

The inverse is 1/3 X $\begin{pmatrix} 9.-3 \\ -11,4 \end{pmatrix}$

= $\begin{pmatrix} 3.-1 \\ -11/3,4/3 \end{pmatrix}$

- Check by multiplying A^{-1} X A.

$\begin{pmatrix} 4,3 \\ 11,9 \end{pmatrix}$ X $\begin{pmatrix} 3.-1 \\ -11/3,4/3 \end{pmatrix}$

- (4×3) + (3×-11/3) = 12 + (-11) = **1**
- (4×-1) + (3×4/3) = -4 + 4 = **0**

- (11×3) + (9×-11/3) = 33 + (-33) = **0**
- (11×-1) + (9×4/3) = -11 + 12 = **1**

= $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

This is the *identity matrix,* confirming that the inverse was correctly calculated.

Listing 9-11 Calculating the inverse of a 2x2 square matrix

```
    .section .data
# [ 4   3 ]
# [ 11  9 ]
# [a b]
# [c d]
matrix:
    .word 4,3,11,9
# Original positions:
# [a b c d]
# Adjugate positions
# [d b c a]
gather_index:
    .word 3,1,2,0
# Print formats
heading:
    .asciz "Inverse matrix:\n"
row:
    .asciz "[ %f  %f ]\n"
.section .text
    .global main
    .extern printf # Use the library for printing
main:
# Prologue
    addi    sp, sp, -32
```

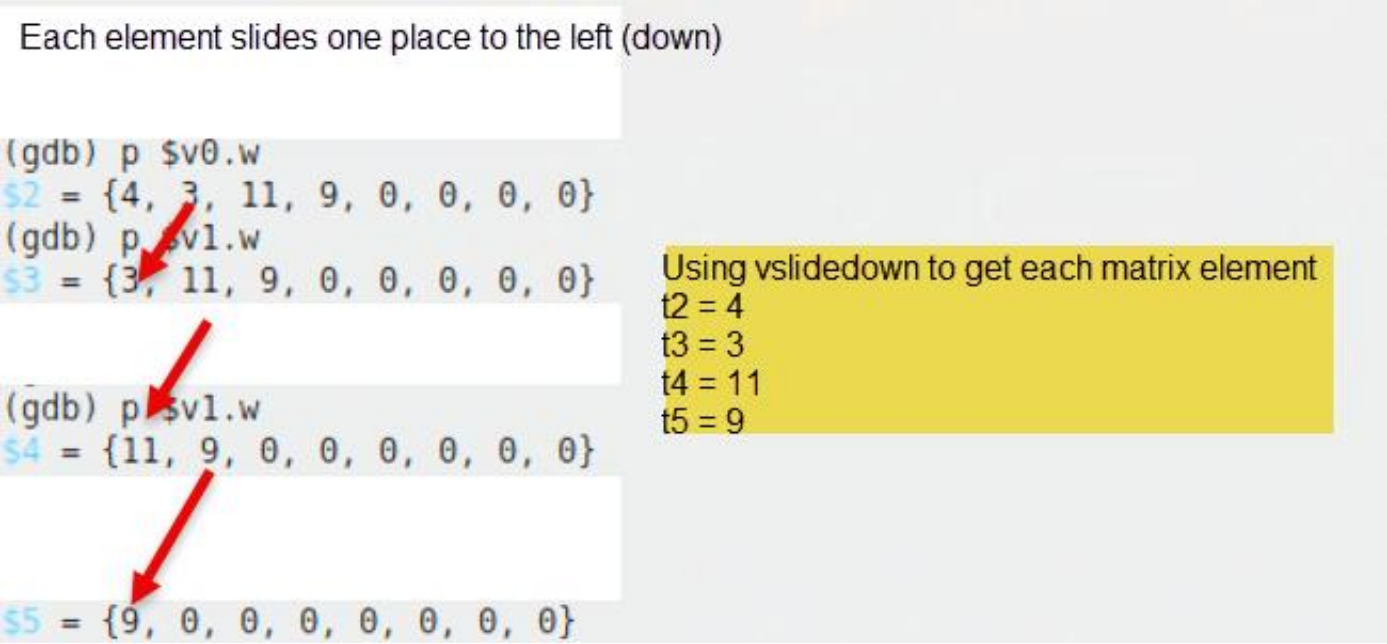
```
    sd      ra, 24(sp)
# Configure vector registers
    li      t0, 4 # 4 matrix members
    vsetvli t0, t0, e32, m1, ta, ma

    la      t1, matrix
    vle32.v v0, (t1)

# v0 = [4 3 11 9]
# Extract a,b,c,d for calculating the determinant
    vmv.x.s t2, v0  # Get a
    vslidedown.vi v1, v0, 1
    vmv.x.s t3, v1 # Get b
    vslidedown.vi v1, v0, 2
    vmv.x.s t4, v1 # Get c
    vslidedown.vi v1, v0, 3
    vmv.x.s t5, v1 # Get d
# determinant = ad - bc
    mul     t6, t2, t5
    mul     s0, t3, t4
    sub     s1, t6, s0
# Now we have the determinant in register S1
# Build adjugate matrix with vrgather
    la      t0, gather_index
    vle32.v v2, (t0)
# v3 = [d b c a]
    vrgather.vv v3, v0, v2
# Extract d,b,c,a values for the adjugate
    vmv.x.s s2, v3 # Get d
    vslidedown.vi v4, v3, 1
    vmv.x.s s3, v4 # Get b

    vslidedown.vi v4, v3, 2
```

```
    vmv.x.s s4, v4 # Get c
    vslidedown.vi v4, v3, 3
    vmv.x.s s5, v4 # Get c
# Negate middle values
    neg     s3, s3
    neg     s4, s4

# Result:
# [ s2  s3 ]
# [ s4  s5 ]
# Convert determinant to float
    fcvt.s.w fa7, s1 # Determinant now in fa7
# Print title
    la      a0, heading
    call    printf

# First row
# Convert to float
    fcvt.s.w fa0, s2
    fcvt.s.w fa1, s3
# Divide by determinant
    fdiv.s  fa0, fa0, fa7
    fdiv.s  fa1, fa1, fa7
# Convert float -> double
    fcvt.d.s fa0, fa0
    fcvt.d.s fa1, fa1
# move doubles into integer registers
    fmv.x.d a1, fa0
    fmv.x.d a2, fa1
    la      a0, row
    call    printf
# Second row
    fcvt.s.w fa0, s4
```

```
    fcvt.s.w fa1, s5
    fdiv.s  fa0, fa0, fa7
    fdiv.s  fa1, fa1, fa7

    fcvt.d.s fa0, fa0
    fcvt.d.s fa1, fa1
    fmv.x.d a1, fa0
    fmv.x.d a2, fa1
    la      a0, row
    call    printf
# Tidy up stack!
    ld      ra, 24(sp)
    addi    sp, sp, 32
    li      a0, 0
    ret
```

Use `gcc -march=rv64gcv -g -o listing9-11 listing9-11.s` to compile!

The output from gdb below illustrates how vslidedown works to extract the elements of the matrix one at a time.



9.6 Strip-mining

The RVV specification refers to strip-mining as a method for handling a large number of elements using loop iterations. Stripmining is used to process more elements than can be provided with a given vector configuration. The `vsetvli` command will request an Application Vector Length (AVL) value, and the destination register will hold a value indicating how many elements can be processed per iteration. Normally it will take a number of passes to process all the elements, since the available quantity is usually less than the requested quantity. After the final pass, any elements that are not used are referred to as tail elements. For example, if 30 elements were requested and eight were granted at a time, the fourth and final pass would require only six elements, and there would be two left unused. As we have seen, we can set up a policy to leave them undisturbed and retain their existing values, or treat them as agnostic where we do not care about their values. This approach is portable in that no assumptions are made regarding the hardware capabilities of the system. The vector length is *dynamic*.

9.6.1.1 Portability

Assume a system has a VLEN value of 256 (VLENB=32), as shown in the diagram. This requires three loops if a total of 24 elements are to be processed with an element size of 32 bits.

Vlen=256 vlenb=32 (BananaPI-BF3)

Vector register 256 bits wide

Element 7	Element 6	Element 5	Element 4	Element 3	Element 2	Element 1	Element 0
32 bits	32 bits	32 bits	32 bits	32 bits	32 bits	32 bits	32 bits
Bits 255:224	Bits 223:192	Bits 191:160	Bits 159:128	Bits 127:96	Bits 95:64	Bits 63:32	Bits 31:0

Now, if the program were to run on a more powerful machine with VLEN=512, it would be able to process 16 elements per iteration rather than 8. No software code changes are required; the program automatically optimizes according to the hardware.

In summary, this is a huge benefit over SIMD-only systems! We do not have to worry about the capabilities of the underlying system. A system with a VLEN of 512 will run twice as efficiently as a node with a VLEN of 256. This is transparent to us when coding, and no special compilation switches are necessary.

The next program uses 8 elements for each vector register.

Listing 9-12 vector loop (VLEN=256, e32)

```
    # Square the first 20 integers using RISC-V Vector Extensions, Squares are stored as words
    .section .data
    integer_array:
        .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
    .section .bss
    .align 4
    array_of_squares:
        .zero 72
```

```
    .equ number_of_elements, 20

    .section .text

    .global _start

    _start:

        la      a0, integer_array

        la      a1, array_of_squares

        li      a2, number_of_elements

    vector_loop:

        vsetvli  t0,  a2,  e32,  m1,  tu,  ma  # Tail elements left undisturbed, t0 holds the granted
number of elements

        vle32.v v1, (a0) # Load up v1 with the elements pointed to by reg a0

        vmul.vv v2, v1, v1 # Square them, putting the result in v2

        vse32.v v2, (a1) # Store the values in v2 into the destination memory pointed to by a1

        sub     a2, a2, t0          # Subtract current vector length (vl) from no of elements left

        slli    t1, t0, 2           # vl * 4, t1 holds increment value

        add     a0, a0, t1          # src incremented by 32 (8 elements x 32 bits)

        add     a1, a1, t1          # dst incremented by 32 (8 32-bit words in memory)

        bnez    a2, vector_loop

    # Three loop iterations, vl=8,vl=8,vl=4

        li      a7, 93

        ecall
```

```
(gdb) x /20wd &array_of_squares
0x11170:        1       4       9       16
0x11180:        25      36      49      64
0x11190:        81      100     121     144
0x111a0:        169     196     225     256
0x111b0:        289     324     361     400
(gdb)
```

The next example reduces the element size to process 16 elements per iteration.

Listing 9-13 vector loop (VLEN=256, e16)

```
# Square first 20 integers using RISC-V Vector Extensions, this example uses halfwords for storage

    .section .data

    integer_array:

        .half  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

    .section .bss

    .align 4
```

```
    array_of_squares:

        .zero 40

    .equ number_of_elements, 20

    .section .text

    .global _start

    _start:

        la      a0, integer_array

        la      a1, array_of_squares

        li      a2, number_of_elements

    vector_loop:

        vsetvli t0, a2, e16, m1, tu, ma # Tail elements left undisturbed, t0 holds the granted
number of elements

        vle16.v v1, (a0) # Load up v1 with the elements pointed to by reg a0

        vmul.vv v2, v1, v1 # Square them, putting the result in v2

        vse16.v v2, (a1) # Store the values in v2 into the destination memory pointed to by a1

        sub     a2, a2, t0    # Subtract current vector length (vl) from number of elements left

        slli    t1, t0, 1     # vl * 4, t1 holds increment value

        add     a0, a0, t1    # src incremented by 32 (16 elements x 16 bits)

        add     a1, a1, t1    # dst incremented by 32 (16 16-bit words in memory)

        bnez    a2, vector_loop # Two iterations vl=16,vl=4

        li      a0, 0

        li      a7, 93

        ecall
```

```
(gdb) i reg vl
vl             0x10     16
(gdb) n
(gdb) i reg vl
vl             0x4      4
(gdb) n
(gdb) x /20dh &array_of_squares
0x11150:        1       4       9       16      25      36      49      64
0x11160:        81      100     121     144     169     196     225     256
0x11170:        289     324     361     400
```

Only two iterations required this time!

It is imperative to refrain from hardcoding the strip-mining value. Instead, the residual count should be supplied to `vsetvli` during each iteration, thereby enabling the system to report the actual number that will be processed. This approach is fundamental to ensuring portability across central processing units (CPUs) equipped with diverse VLEN values.

9.6.1.2 Stride

A **stride** is defined as the byte-wise distance between consecutive memory elements during vector data loading or storage. Previously, the default stride value of 1 has been utilized with the standard vector load (vle) and vector store (vse) instructions. The strided load and store instructions exhibit a similar format:

```
vlse<element size in bits> <vector destination register>, <base memory address><Stride amount in bytes>
```

An example is `vlse32.v, v1, (a0), t1`

Stride values are used with:

- `vlse*` instructions → vector load strided elements.
- `vsse*` instructions → vector store strided elements.

An example follows:

This code uses a stride value of two. A value of two causes the system to process every second element.

The earlier examples used a default *unit stride* for processing. The instruction `vlse32.v v1, (a0), t1` fetches the stride count from register t1 and loads it from the memory location pointed to by register a0. The `vse32.v v2, (a1)` stores every second value from the stride load command into the memory pointed to by register a1. It will continue the loop until all elements have been processed.

Listing 9-14 Stripmining with a stride value of 2

```
# Illustrates the use of stride=2, processes every second number
.data
.balign 4
# Input array: 1, 2, 3, ... up to 40 (to ensure we have 20 elements at stride 2)
inputdata:  .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
            .word 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40
.balign 4
outputdata: .zero 80   # Space for 20 squared integers (20 * 4 bytes)
.equ stridecount, 8 # (2 by 4 (word)
.equ elements_to_process, 20
.text
.global _start
_start:
    la a0, inputdata    # Source base address
    la a1, outputdata   # Destination base address
    li t0, elements_to_process # Total number of elements to process (n = 20)
    li t1, stridecount  # Byte stride (2 elements * 4 bytes/element)
```

```
loop:
    # Set vector configuration: 32-bit elements, 1 register group (m1)
    # t0 = # of elements requested, t2 receives the number of elements available for the strip
    vsetvli t2, t0, e32, m1, ta, ma
    # Strided load operation(vlse32.v)
    # Loads every 2nd element from (a0) into vector register v1
    vlse32.v v1, (a0), t1
    # Vector Multiplication
    # v2 = v1 * v1 (Squaring)
    vmul.vv v2, v1, v1
    # Strided store operation (vse32.v)
    # Note this could easily be modified to store using every second location
    # Stores results contiguously into location pointed to by a1
    vse32.v v2, (a1)
    # Update Pointers and Counters
    sub t0, t0, t2 # Decrement remaining count by # of elements granted earlier
    # Calculate address offsets:
    # Source moves by (elements processed * stride_in_bytes (vl*t1))
    mul t3, t2, t1
    add a0, a0, t3
    # Destination moves by (elements processed * 4 bytes)
    slli t4, t2, 2
    add a1, a1, t4
    bnez t0, loop        # Continue if elements remain
    li a7, 93
    ecall
```


After the first pass, V1 and V2 contain:

V1 [1,3,5,7,9,11,13,15]

V2 [1,9,25.49.81,121,169,225]

Register A0 points to the original base address +0x40. This is because there were eight words processed (=32 bytes)and we skipped every second one, giving a total of 64 bytes or 0x40. In this case, the initial value set in A0 was 0x11128 and it now holds 0x11168.

Register A1 holds the original address plus an increment of 0x20 since we stored the vector contents into contiguous memory locations.

Register T0 contains 12_{10} since 8 elements have been processed out of the original 20.

Exercises for chapter 9

1. Multiply the inverse matrix obtained in the last listing by its original value. Confirm by obtaining the identity matrix.

9.6.2 Summary of RISC-V Vector Instructions Used in Chapter 9

9.6.2.1 Vector Configuration Instructions

Instruction	Purpose
vsetvli	Configure vector length/type dynamically
vsetivli	Immediate form of vector configuration

9.6.2.2 Vector Load / Store Instructions

Instruction	Purpose
vle8.v	Load 8-bit vector elements
vle16.v	Load 16-bit vector elements
vle32.v	Load 32-bit vector elements
vse16.v	Store 16-bit vector elements
vse32.v	Store 32-bit vector elements
vlse32.v	Strided 32-bit vector load

9.6.2.3 Vector Arithmetic Instructions

Instruction	Purpose
vadd.vv	Vector + vector
vadd.vx	Vector + scalar register
vadd.vi	Vector + immediate
vsub.vv	Vector subtraction
vmul.vv	Vector multiplication
vmul.vx	Vector × scalar
vfadd.vf	Floating-point vector + scalar FP

9.6.2.4 Vector Move / Broadcast Instructions

Instruction	Purpose
vmv.v.x	Broadcast scalar register into vector
vmv.v.i	Fill vector with immediate
vmv.x.s	Move first vector element to scalar register

9.6.2.5 Vector Index / Permutation Instructions

Instruction	Purpose
vid.v	Generate element indices
vslideup.vi	Slide elements upward
vslidedown.vi	Slide elements downward
vrgather.vi	Gather using immediate index
vrgather.vx	Gather using scalar register index
vrgather.vv	Gather using vector index list

9.6.2.6 Vector Mask Instructions

Instruction	Purpose
vmerge.vvm	Merge vectors using mask
vmsne.vi	Vector mask set if not equal
vmnot.m	Invert mask bits (mentioned in comments)

9.6.2.7 Register and CSR Usage

- Vector registers used: v0–v31
- CSR-related instructions include:
    - Reading vector control/status via `csrr` (e.g., csrr t0, vtype)

10 Spike Simulator and Cross-Compiling

Overview of the chapter

Chapter 10 focuses on cross-compiling, which is the process of building RISC-V programs on a non-native host machine such as X86-64 to run on a different architecture (RISC-V). The official RISC-V simulator - Spike will be used to run the cross-compiled programs. Spike supports both 32-bit and 64-bit base ISAs, with support for vector extensions. There is a proxy kernel, `PK` which provides a run-time environment. Spike also supports debugging operations.

The target machine that was used to host Spike is an X86_64 Virtual machine running Ubuntu 24.10 with a username of `ubuntuuser.`

If **not** using precompiled binaries for the toolchain, refer to the following section, which discusses how to build the software.

10.1 Building the Toolchain and Spike

The commands below will be executed during the installation; there are three stages:

- Install and build the RISC-V toolchain
- Install and build Spike
- Install and build PK

```
# Prepare paths, directories, and ownerships

sudo apt update
```

10.1.1Installing the toolchain

Execute the following commands -

```
$ sudo apt-get install device-tree-compiler autoconf automake autotools-dev curl python3 python3-pip libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-dev binutils gcc libpthread-stubs0-dev libboost-all-dev

mkdir ~/riscv

cd ~/riscv

sudo mkdir /opt/riscv

sudo chown ubuntuser:ubuntuser /opt/riscv

echo 'export PATH=/opt/riscv/bin:$PATH' >> ~/.bashrc

source ~/.bashrc


# Clone from Github
```

```
git clone https://github.com/riscv/riscv-gnu-toolchain

git clone https://github.com/riscv-software-src/riscv-isa-sim

git clone https://github.com/riscv-software-src/riscv-pk


# Build toolchain

cd ~/riscv/riscv-gnu-toolchain

mkdir build && cd build

../configure --prefix=/opt/riscv

make -j$(nproc)

# Check

riscv64-unknown-elf-gcc -v

# Build C program

# include <stdio.h>

int main ()

{

        printf ("Hello RISC-V!\n");

        return 0;

}

riscv64-unknown-elf-gcc ~/helloriscv.c
```

Note that the resulting `a.out file` is not executable on the host machine!

10.1.2Installing Spike

```
Build Spike

cd ~/riscv/riscv-isa-sim

mkdir build && cd build

../configure --prefix=/opt/riscv

make -j$(nproc)

sudo make install

# Check

spike --help
```

10.1.3Installing PK

```
#Build PK

cd ~/riscv/riscv-pk

mkdir build && cd build
```

```
../configure --prefix=/opt/riscv --host=riscv64-unknown-elf

make -j$(nproc)

sudo make install
```

#Check

Use `a.out` (left over from the c compilation above).

Execute the file within the Spike environment.

```
$ spike pk ./a.out

Hello, RISC-V!
```

Examine the file type

```
$ readelf -h a.out

ELF Header:

  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00

  Class:                             ELF64

  Data:                              2's complement, little endian

  Version:                           1 (current)

  OS/ABI:                            UNIX - System V

  ABI Version:                       0

  Type:                              EXEC (Executable file)

  Machine:                           RISC-V

  Version:                           0x1

  Entry point address:               0x1014e

  Start of program headers:          64 (bytes into file)

  Start of section headers:          22816 (bytes into file)

  Flags:                             0x5, RVC, double-float ABI

  Size of this header:               64 (bytes)

  Size of program headers:           56 (bytes)

  Number of program headers:         4

  Size of section headers:           64 (bytes)

  Number of section headers:         15

  Section header string table index: 14
```

The program can be transferred to a native RISC-V host and executed on that host. In the example below, the file has been transferred to a Banana Pi BF3 RISC_V native host and then executed.

```
$ scp a.out 192.168.68.231:
user@192.168.68.231's password:
a.out
$ ./a.out
Hello, RISC-V!
```

Typically, programmers use Spike for initial development and then test their final releases on a native target host.

10.2 Cross-assembling and linking

The command line for assembling and linking is similar to the commands that run on a native host. To differentiate the RISC_V tools from the (in this case) X86-64 tools, they are preceded here with `riscv64-unknown-elf-<toolname>.`

Writing the `HelloRiscv` program followed by cross-assembling and linking in pure assembly, is shown below –

```
$ cat hellorisc.s
# hellorisc.s
.section .text
.global _start
_start:
li a0, 1 # use a0 for stdout
la a1, message # Load the address of the message text
li a2, 12 # Store the message length
li a7, 64     # Write syscall
ecall

li a7, 93  # Exit syscall
ecall
.data
message: .ascii "Hello RISCV\n"
$ riscv64-unknown-elf-as -g -o hellorisc.o hellorisc.s
$ riscv64-unknown-elf-ld -o hellorisc hellorisc.o
$ spike --isa=rv64gcv pk hellorisc
Hello RISCV
```

10.2.1 Using objdump

The command to dump the executable is-

```
riscv64-unknown-elf-objdump -d helloriscv.
```

The disassembled `.text` section looks like:

```
000000000001014e <_start>:
   1014e:     00003197    auipc   gp,0x3
   10152:     6ca18193    addi    gp,gp,1738 #13818 <__global_pointer$>
   10156:     00004517    auipc   a0,0x4
   1015a:     86250513    addi    a0,a0,-1950 # 139b8 <__stdio_exit_handler>
   1015e:     00004617    auipc   a2,0x4
   10162:     e1a60613    addi    a2,a2,-486 # 13f78 <__BSS_END__>
   10166:     8e09        sub     a2,a2,a0
   10168:     4581        li      a1,0
   1016a:     742000ef    jal     108ac <memset>
   1016e:     00001517    auipc   a0,0x1
   10172:     95450513    addi    a0,a0,-1708 # 10ac2 <atexit>
   10176:     c519        beqz    a0,10184 <_start+0x36>
   10178:     00002517    auipc   a0,0x2
   1017c:     ac250513    addi    a0,a0,-1342 # 11c3a <__libc_fini_array>
   10180:     143000ef    jal     10ac2 <atexit>
   10184:     6c6000ef    jal     1084a <__libc_init_array>
   10188:     4502        lw      a0,0(sp)
   1018a:     002c        addi    a1,sp,8
   1018c:     4601        li      a2,0
   1018e:     04e000ef    jal     101dc <main>
   10192:     b779        j 10120 <exit>
```

For more information on Spike and the proxy kernel, refer to the resources listed at the end of this chapter.

10.3 Final comments

This book is focused on presenting RISC-V assembly language under a Linux operating system environment. It has not covered “bare-metal coding” which involves more complexity. There are some resources given at the end of this chapter.

Spike is more useful in this situation where a host operating system is not available. For those of us using native RISC-V SBCs, Spike is less relevant.

Companion video tutorial links can be found at risccomputing.com

10.3.1 Further Resources

- Information steps for Spike installation can be found at GitHub:

https://github.com/riscv-software-src/riscv-isa-sim).

- Installation steps for the RISC-V toolchain can be found at GitHub:

https://github.com/riscv-collab/riscv-gnu-toolchain)

- Risc-V toolchain projects:

https://riscv.atlassian.net/wiki/spaces/HOME/pages/16154663/Toolchain+Projects

- Bare metal coding tutorial:

Bare metal programming with RISC-V guide

Writing a bare-metal RISC-V application in D | Zachary Yedidia's blog

The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.

GDB Commonly Used Commands

Command	Description	Example
(B)reak	Set breakpoint	"b _start" "b 1"
	Conditional break	Break myloop if $t0 == 36
(D)elete	Delete breakpoints	"d" followed by "y"
(I)nfo b	Show breakpoints	"i b"
(I)nfo (ad)dress	Show the location of a Symbol	"i ad _start"
(I)nfo files	Show the names of files being debugged	"i files"
(I)nfo (R)egisters	List the integer registers	"i r "
(I)nfo (R)egister n	List the content of an individual register	"i r t0"
(I)nfo (R)egisters (V)ector	Shows vector-related registers	" i r v"
(I)nfo (R)egisters CSR	Shows control and status registers	"I r csr"
P $vn.w	Prints the vector register Vn as groups of words	"p $v2.w"
P (int[m]) $vn	Prints all word elements of a vector	"p (int[8]) $v1
P (short[m]) $vn	Prints all short elements of a vector	"p (short [16]) $v1
P /t $register	Print values in bit format	"p /t $a1"
p ((int[8])$vn)[m]	Print a single array element of a vector interpreted as 32-bit ints	"p ((int[8])$v0)[3]"
p /t ((int[8])$vn)[m]	Print a single array element of a vector interpreted as 32-bit ints	"p /t ((int[8])$v0)[0]"
(I)nfo source	Info about the source file being debugged	"i source"
(I)nfo symbol &_start	Show the section location of a symbol	" i symbol _start"
(I)nfo (va)riables	Shows addresses of variables	" I ((va"
(I)nfo win	Shows windows used in TUI	"i win"
(Main)tenance (i)nfo (t)arget-sections	Shows section information	"mai i t#
N(ext)	Steps n lines (default is 1) and steps over a sub-routine	"n" "n 3"
S(tep)	Steps n lines (default is 1) and steps into a sub-routine	"s" "s 2"

Appendix A

TUI reg N(ext)	Shows the next set of registers	"tui reg n"
TUI reg vec	Shows the vector registers	"tui reg vec"
Tui reg all	Shows all registers	"tui reg all"
x/FMT address	Shows # of memory locations (n), format (f) such as x(hex), d(decimal), f(float), s(string) and size such as b(byte), h(halfword), w (word), g (giant 8 bytes)	X /2xg 0x11100
x/FMT register	Shows # of memory locations when a register holds an address (n), format (f) such as x(hex), d(decimal), f(float), s(string) and size such as b(byte), h(halfword), w (word), g (giant 8 bytes)	X /2xg $SP
x/ni $pc	Disassemble the next n instructions from the program counter	x/2i $pc
Up and down arrow	Cycles through commands, use <Ctrl> P(revious) or <Ctrl> N(ext) if using the TUI	
Refresh	<Ctrl-L>	

Enable TUI

The TUI can be enabled by default by adding the following lines to `~/.gdbinit`

```
Layout split
Layout regs
Set history save on
Set history filename ~/gdbhistory
Set logging enabled on
```

Print format commands

GDB print Command	**Explanation**
p/x	hexadecimal
p/d	signed decimal
p/u	unsigned decimal
p/o	octal
p/t	binary
p/c	character

Appendix B

GDB

p/f floating point

Examples:

```
P/t $v3.w
```
Prints the word contents in binary format

{0,0,0,0,1010, 10100, 11110, 101000}

```
P/u $v3.w
```
Prints the word contents in unsigned decimal format.

{0,0,0,0,10,20,20,40)

```
P/x $sp
```
Prints out contents of stack pointer in hex

0x3fffffeb80

Appendix B

ASCII Code

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0x00	Null character	32	0x20	SPACE	64	0x40	@	96	0x60	`
1	0x01	Start of heading	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	Start of text	34	0x22	”	66	0x42	B	98	0x62	b
3	0x03	End of text	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	End of transmission	36	0x24	$	68	0x44	D	100	0x64	d
5	0x05	Enquiry	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	Acknowledgment	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	Bell	39	0x27	’	71	0x47	G	103	0x67	g
8	0x08	Backspace	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	Horizontal tab	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	Line feed	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	Vertical tab	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0x0C	Form feed	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0x0D	Carriage return	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0x0E	Shift out	46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0x0F	Shift in	47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10	Data link escape	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	Device Control 1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	Device Control 2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	Device Control 3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	Device Control 4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	Negative Acknowledgment	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	Synchronous Idle	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	End of Transmission Block	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	Cancel	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	End of Medium	57	0x39	9	89	0x59	Y	121	0x79	y

Appendix B

ASCII Codes

26	0x1A	Substitute	58	0x3A	:	90		0x5A	Z	122	0x7A	z
27	0x1B	Escape	59	0x3B	;	91		0x5B	[	123	0x7B	{
28	0x1C	File Separator	60	0x3C	<	92		0x5C	|	124	0x7C	
29	0x1D	Group Separator	61	0x3D	=	93		0x5D	]	125	0x7D	}
30	0x1E	Record Separator	62	0x3E	>	94		0x5E	^	126	0x7E	~
31	0x1F	Unit Separator	63	0x3F	?	95		0x5F	_	127	0x7F	<DEL>

Appendix B

Instruction Set

RV32I / RV64I Base Integer ISA

Arithmetic Instructions

Mnemonic	Description
ADD	rd = rs1 + rs2
sub	rd = rs1 − rs2
addi	rd = rs1 + imm
lui	Load upper immediate
auipc	PC-relative upper immediate
slt / SLTU	Set < (signed/unsigned)
slti / SLTIU	Immediate versions

RV64-only arithmetic

Mnemonic	Description
addw	32-bit ADD, sign-extend
subw	32-bit subtract, sign-extend
addiw	32-bit ADD immediate

Logical Instructions

Mnemonic	Description
AND / OR / XOR	Bitwise ops
ANDI / ORI / XORI	Immediate versions
SLL / SRL / SRA	Shifts
SLLI / SRLI / SRAI	Shift immediates

Shift immediate width:

RV32: 5-bit shamt

RV64: 6-bit shamt

Control-Flow Instructions

Mnemonic	Description

JAL	Jump and link
JALR	Indirect jump
BEQ / BNE	Branch equal/not
BLT / BGE	Signed branches
BLTU / BGEU	Unsigned branches

Load Instructions

Mnemonic	Description
LB / LBU	Load byte
LH / LHU	Load halfword
LW	Load word
LWU	Load word unsigned
LD	Load doubleword

Store Instructions

Mnemonic	Description
SB	Store byte
SH	Store halfword
SW	Store word
SD	Store doubleword

System Instructions

Mnemonic	Description
ECALL	Environment call
EBREAK	Breakpoint
FENCE	Memory fence
FENCE.I	Instruction fence
CSRRW / CSRRS / CSRRC	CSR ops
CSRRWI / CSRRSI / CSRRCI	Immediate CSR ops

M Extension (Multiply/Divide)

Mnemonic	Description
MUL	Low 32/64-bit product

MULH	High signed×signed
MULHSU	High signed×unsigned
MULHU	High unsigned×unsigned
DIV / DIVU	Signed/unsigned divide
REM / REMU	Signed/unsigned remainder

RV64-only

Mnemonic	**Description**
MULW	32-bit product
DIVW / DIVUW	32-bit divide
REMW / REMUW	32-bit remainder

A Extension (Atomics)

Mnemonic	**Description**
LR.W / LR.D	Load-reserved
SC.W / SC.D	Store-conditional
AMOSWAP.W/D	Atomic swap
AMOADD.W/D	Atomic ADD
AMOXOR.W/D	Atomic xor
AMOAND.W/D	Atomic and
AMOOR.W/D	Atomic or
AMOMIN.W/D	Atomic min (signed)
AMOMAX.W/D	Atomic max (signed)
AMOMINU.W/D	Atomic min (unsigned)
AMOMAXU.W/D	Atomic max (unsigned)

RV32 uses *.W* only; RV64 supports both *.W* and *.D*.

F Extension (Single-Precision FP)

Mnemonic	**Description**
FLW / FSW	Load/store float
FADD.S	Add

Appendix C

FSUB.S	Sub
FMUL.S	Mul
FDIV.S	Div
FSQRT.S	Sqrt
FSGNJ.S / FSGNJN.S / FSGNJX.S	Sign ops
FMIN.S / FMAX.S	Min/max
FMADD.S / FMSUB.S / FNMSUB.S / FNMADD.S	FMA
FCVT.W.S / FCVT.WU.S	FP→int
FCVT.S.W / FCVT.S.WU	Int→FP
FEQ.S / FLT.S / FLE.S	Compare
FCLASS.S	Classify
FMV.X.W / FMV.W.X	Bit moves

D Extension (Double-Precision FP)

Mnemonic	Description
FLD / FSD	Load/store double
FADD.D	Add
FSUB.D	Sub
FMUL.D	Mul
FDIV.D	Div
FSQRT.D	Sqrt
FSGNJ.D / FSGNJN.D / FSGNJX.D	Sign ops
FMIN.D / FMAX.D	Min/max
FMADD.D / FMSUB.D / FNMSUB.D / FNMADD.D	FMA
FCVT.W.D / FCVT.WU.D	FP→int
FCVT.D.W / FCVT.D.WU	Int→FP
FCVT.S.D / FCVT.D.S	FP↔FP
FEQ.D / FLT.D / FLE.D	Compare
FCLASS.D	Classify

Appendix C

FMV.X.D / FMV.D.X	Bit moves

C Extension (Compressed)

16-bit encodings

Mnemonic	Expands To
C.addi	addi
C.ADD	ADD
C.addw	addw (RV64 only)
C.LI	LI
C.lui	lui
C.MV	MV
C.J	JAL x0
C.JR	JALR x0
C.JAL	JAL x1
C.BEQZ / C.BNEZ	BEQ/BNE
C.LW / C.SW	LW/SW
C.LD / C.SD	LD/SD (RV64 only)
C.AND / C.OR / C.XOR	Logical ops
C.SLLI	SLLI
C.SRAI / C.SRLI	Shift immediates
C.NOP	addi x0, x0, 0

V Extension (RVV 1.0 Vector ISA)

Vector Configuration

Mnemonic	Description
VSETVLI	Set VL/VTYPE
VSETVL	Set VL only

Vector Loads/Stores

Category	Mnemonics
Unit-stride	VLx, VSx

Strided	VLSx, VSSx
Indexed	VLUX, VLOX, VSUX, VSOX
Whole register	VL1R, VL2R, VL4R, VL8R

Integer Vector Arithmetic

Category	**Mnemonics**
Add/Sub	VADD, VSUB, VRSUB
Multiply	VMUL, VMULH, VMULHSU, VMULHU
Divide	VDIV, VDIVU, VREM, VREMU
Widening	VWADD, VWSUB, VWMUL, VWMULU, VWMULSU
Narrowing	VNCLIP, VNCLIPU

Logical / Bit Ops

Category	**Mnemonics**
Logic	VAND, VOR, VXOR
Shifts	VSLL, VSRL, VSRA
Bit ops	VPOPC, VFIRST, VMSBF, VMSIF, VMSOF

Vector Mask Ops

Category	**Mnemonics**
Mask logic	VMAND, VMOR, VMXOR, VMNAND, VMNOR, VMXNOR
Mask set/clear	VMSNE, VMSEQ, VMSLT, VMSLE, VMSGT

Floating-Point Vector Arithmetic

Category	**Mnemonics**
Add/Sub	VFADD, VFSUB
Mul/Div	VFMUL, VFDIV
FMA	VFMACC, VFNMACC, VFMADD, VFNMADD
Compare	VFEQ, VFLT, VFLE
Convert	VFCVT.X.F, VFCVT.F.X, VFCVT.F.F

Appendix C

Classify	VFCLASS

Vector Reductions

Category	Mnemonics
Integer	VREDSUM, VREDMAX, VREDMIN, VREDMAXU, VREDMINU
FP	VFREDOSUM, VFREDUSUM, VFREDMAX, VFREDMIN

Category	Mnemonics
Slide	VSLIDEUP, VSLIDEDOWN
Gather/Scatter	VRGATHER, VRGATHEREI16
Compress	VCOMPRESS

Pseudoinstructions

Register-move and constant construction

Pseudoinstruction	Expansion
nop	addi x0, x0, 0
mv rd, rs	addi rd, rs, 0
not rd, rs	xori rd, rs, -1
neg rd, rs	sub rd, x0, rs
negw rd, rs (RV64)	subw rd, x0, rs
li rd, imm	expands to addi or lui/addi sequence
la rd, symbol	auipc + addi (or lui/addi depending on model)

Load/store pseudoinstructions

Pseudo	Expansion
lb rd, symbol	lui + lb
lh rd, symbol	same pattern
lw rd, symbol	same pattern
ld rd, symbol (RV64)	same pattern

sb rs, symbol	lui + sb
sh rs, symbol	same pattern
sw rs, symbol	same pattern
sd rs, symbol (RV64)	same pattern

Control-flow pseudoinstructions

Pseudo	Expansion
j label	jal x0, label
jr rs	jalr x0, 0(rs)
ret	jalr x0, 0(ra)

Set-less-than pseudoinstructions

Pseudo	Expansion
seqz rd, rs	sltiu rd, rs, 1
snez rd, rs	sltu rd, x0, rs
sltz rd, rs	slt rd, rs, x0
sgtz rd, rs	slt rd, x0, rs

Bit-test pseudoinstructions

Pseudo	Expansion
beqz rs, label	beq rs, x0, label
bnez rs, label	bne rs, x0, label
blez rs, label	bge x0, rs, label
bgez rs, label	bge rs, x0, label
bltz rs, label	blt rs, x0, label
bgtz rs, label	blt x0, rs, label

RV64 sign-extension pseudoinstructions

Pseudo	Expansion
sext.w rd, rs	ADDIW rd, rs, 0

Shift pseudoinstructions

Pseudo	Expansion
slli rd, rs, shamt	assembler allows symbolic shamt
srli rd, rs, shamt	
srai rd, rs, shamt	

Compressed aliases

Alias	Real instruction
c.mv	c.ADD with rs2 ≠ x0
c.jr	c.jalr with rd = x0
c.nop	c.addi x0, 0

Vector pseudoinstructions

Pseudo	Expansion
vmv.v.i vd, imm	vaddi.vi vd, v0, imm
vmv.v.x vd, rs	vadd.vx vd, v0, rs

Appendix C

References and Resources

- Green Card Reference Sheet
- https://cs315-f24.cs.usfca.edu/files/RISCVGreenCardv8.pdf
- Ratified Specifications
- https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications
- RISC-V Summits https://riscv.org/community/risc-v-summits/

Appendix D

Assembly Directives

.text	Beginning of the code (text) section.
.data	Beginning of the data section.
.bss	Beginning of the uninitialized data section.
.rodata	Beginning of read-only section
.global or .globl	Declares a symbol as global, making it accessible across files.
.section	Specifies a named section.
.align	Aligns the next item to a specified boundary.
.byte	Allocates and initializes 1-byte data
.half/.2byte	Allocates and initializes 2-byte data.
.word/.4byte	Allocates and initializes 4-byte data.
.dword/.8byte	Allocates and initializes 8-byte data.
.string or .asciz	Allocates string space with null termination.
.ascii –	Allocates string space without a null terminator.
.space N	Reserves a specified number of bytes without initialization.
.zero N	Reserves and zeroes a specified number of bytes.
.equ or .set	Defines a constant value for a symbol.
.type	Specifies the symbol type.
option arch,rv64imafdc	-Specify ISA.
.option pic / .option nopic	Position-independent code mode.
.option relax,/mno-relax	Relaxation

INDEX