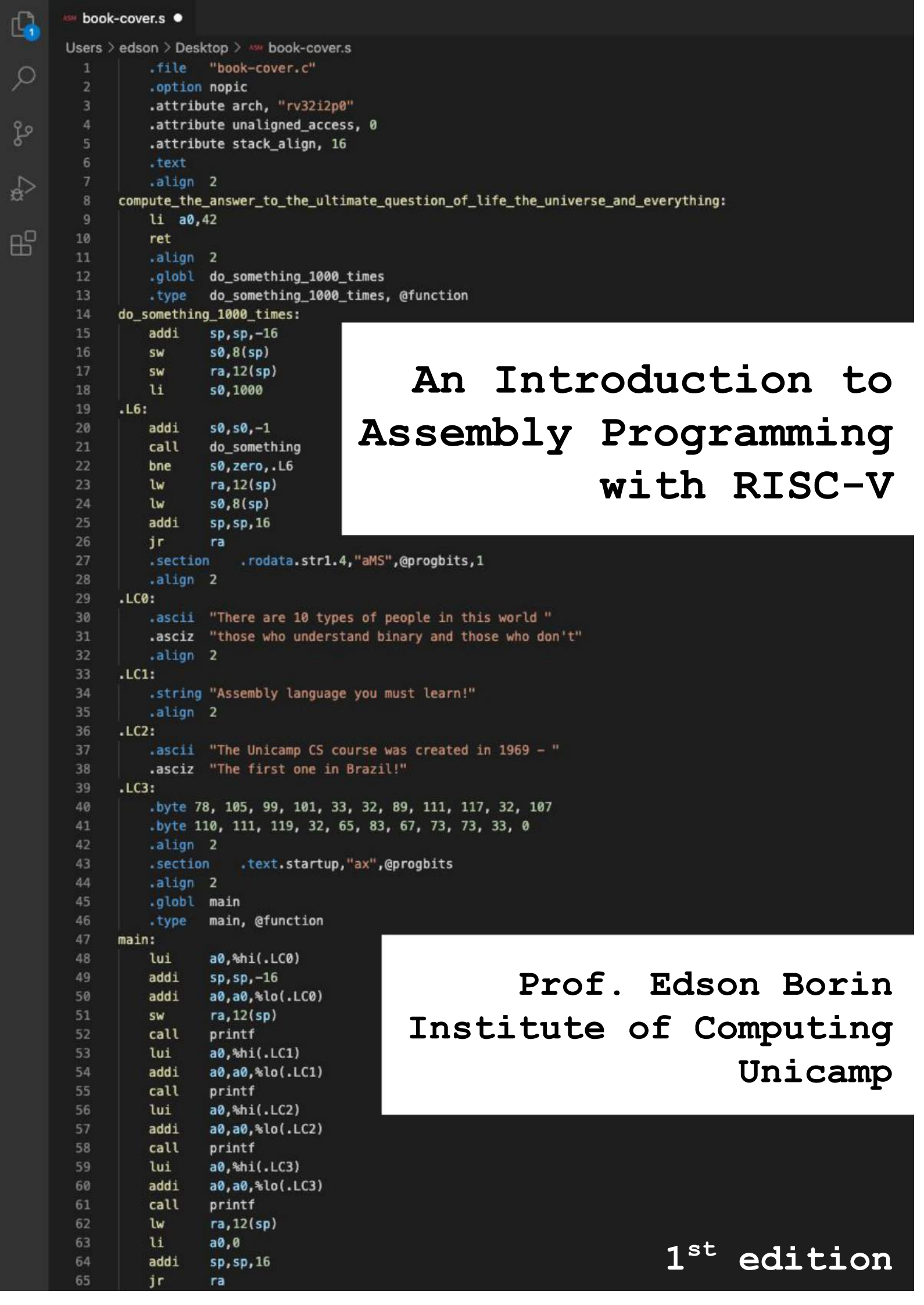


An Introduction to Assembly Programming with RISC-V

Prof. Edson Borin
Institute of Computing
Unicamp

1st edition

An Introduction to Assembly Programming
with RISC-V

ISBN:978-65-00-15811-3
First edition 2021

Edson Borin
Institute of Computing - University of Campinas
Av. Albert Einstein, 1251
Cidade Universitária Zeferino Vaz
Barão Geraldo - Campinas - SP - Brasil
`www.ic.unicamp.br/~edson`
13083-852

An updated version of this book and other material may be available at:
`www.riscv-programming.org`

Foreword

This book focuses on teaching the art of programming in assembly language, using the RISC-V ISA as the guiding example. Towards this goal, the text spans, at an introductory level, the organization of computing systems, describes the mechanics of how programs are created and introduces basic programming concepts including both user level and system programming. The ability to read and write code in low-level assembly language is a powerful skill to be able to create high performance programs, and to access features of the machine that are not easily accessible from high-level languages such as C, Java or Python, for example to control peripheral devices.

The book introduces the organization of computing systems, and the mechanics of creating programs and converting them to machine-readable format suitable for execution. It also teaches the components of a program, or how a programmer communicates her intent to the system via directives, data allocation primitives and finally the ISA instructions, and their use. Basic programming concepts of control flow, loops as well as the runtime stack are introduced.

Next the book describes the organization of code sequences into routines and subroutines, to compose a program. The text also addresses issues related to system programming, including notions of peripheral control and interrupts.

This text, and ancillary teaching materials, has been used in introductory classes at the University of Campinas, Brazil (UNICAMP) and has undergone refinement and improvement for several editions.

Mauricio Breternitz
Principal Investigator & Invited Associate Professor
ISTAR ISCTE Laboratory
ISCTE Instituto Universitario de Lisboa
Lisbon, Portugal

Notices:

- Document version: May 9, 2022
- Please, report typos and other issues to Prof. Edson Borin (`edson@ic.unicamp.br`).

Contents

Glossary

32-bit address space is set of addresses represented by 32-bit unsigned numbers. 10, 54

binary digit is a digit that may assume one of two values: “0” (zero) or “1” (one). 10, 11, 14

bus is a communication system that transfers information between the computer components. This system is usually composed of wires that are responsible for transmitting the information and associated circuitry, which are responsible for orchestrating the communication. 3, 10, 107–109, 112

byte addressable memory is a memory in which each memory word stores a single byte. 3–5, 10, 18–21, 23, 54

Central Processing Unit , or CPU, is the computer component responsible for executing the computer programs. 2, 3, 10, 13

column-major order specifies that the elements of a two-dimensional array are organized in memory column by column. In this context, the elements of the first column are placed first then the elements of the second column are placed after the elements of the first one and so on. 10

Control and Status Register , or CSR, is an internal CPU register that exposes the CPU status to the software and allow software to control the CPU. 10, 120, 121, 129, 131

endianness refers to the order in which the bytes are stored on a computing system. There are two common formats: little-endian and big-endian. The little-endian format places the least significant byte on the memory position associated with the lowest address while the big-endian format places the most significant byte on the memory position associated with the highest address. 10, 19, 46, 64–67

exceptions are events generated by the CPU in response to exceptional conditions when executing instructions. 10

external interrupts are interrupts caused by external (non-CPU) hardware, such as peripherals, to inform the CPU they require attention. 10

hardware interrupts are events generated by hardware, such as peripherals, to inform the CPU they require attention. 10

immediate value is a number that is encoded into the instruction encoding. As a consequence, it is a constant. 10, 57–69, 72, 135

Instruction Set Architecture defines the computer instructions set, including, but not limited to, the behavior of the instructions, their encoding, and resources that may be accessed by the instructions, such as CPU registers. 4, 10, 49, 53, 54, 56, 60, 62, 64–67, 70–72, 74, 75, 84, 120, 127, 128

integer overflow occurs when the result of an arithmetic operation on two integer m-bit binary numbers is outside of the range that can be represented by an m-bit binary number. 10, 15, 16

interrupt service routine , or ISR, is a software routine that handles interrupts. It is also nown as interrupt handler. 10, 117, 118, 121–125, 128, 129, 131, 132

interrupt vector table is a table that maps interrupt/exception identifiers to routines that must be invoked to handle the interrupt/exception. The interrupt vector table is usually stored in main memory and accessed by the CPU hardware to invoke the proper routine when handling an interrupt/exception. 10, 131

ISA native datatype is a datatype that can be naturally processed by the ISA. 10, 54, 64

load instruction is an instruction that loads a value from main memory into a register. 10, 56

Load/Store architecture is a computer architecture that requires values to be loaded/stored explicitly from/to main memory before operating on them. 10, 55, 56

machine language is a low-level language that can be directly processed by the computer's central processing unit (CPU). 10, 25, 26

main memory is a storage device used to store the instructions and data of programs that are being executed. 2–5, 10, 12, 18, 20, 21, 23, 27, 32, 33, 35, 36, 48, 107–112, 114–119, 123–125, 128

native program is a program encoded using instructions that can be directly executed by the CPU, without help from an emulator or a virtual machine. 2, 10, 24, 26

numeral system is a system used for expressing numbers. 6–12

opcode the opcode, or operation code, is a code (usually encoded as a binary number) that indicates the operation that an instruction must perform. 10, 57

peripherals are input/output, or I/O, devices that are connected to the computer. Examples of peripheral devices include video cards (also known as graphics cards), USB controllers, network cards, *etc.*. 2, 10, 107

persistent storage is a storage device capable of preserving its contents when the power is shut down. Hard disk drives (HDDs), solid state drives (SSDs), and flash drives are example of persistent storage devices. 2, 3, 10, 107

positional numeral system is a numeral system in which the value of a digit d_i depends on the its position on the sequence. 7, 8, 10, 11

privilege level defines which ISA resources are accessible by the software being executed. 10, 120, 127, 128

privilege mode defines the privilege level for the currently executing software. 10, 13, 128–131

program counter or PC, is the register that holds the address of the next instruction to be executed. In other words, it holds the address of the memory position that contains the next instruction to be executed. It is also known as instruction pointer, or IP, in some computer architectures. 10, 55

pseudo-instruction is an assembly instruction that does not have a corresponding machine instruction on the ISA, but can be translated automatically by the assembler into one, or more, alternative machine instructions to achieve the same effect. 10, 39, 40, 56, 58, 68, 69

register is a small memory device usually located inside the Central Processing Unit (CPU) for quick read and write access. 3, 10

row-major order specifies that the elements of a two-dimensional array are organized in memory row by row. In this context, the elements of the first row are placed first then the elements of the second row are placed after the elements of the first one and so on. 10, 21

stack pointer is a pointer that points to the top of the program stack. In other words, it holds the address of the top of the program stack. In RISC-V, the stack pointer is stored by the `sp` register.. 10

store instruction is an instruction that stores values into main memory. 10

unprivileged ISA is the sub-set of the ISA that is accessible by the software running on unprivileged mode. 10, 55, 128

unprivileged mode is the privilege mode with least privileges. In RISC-V, it is the User/Application privilege mode. 10, 13, 128

unprivileged registers are a set of registers accessible on the unprivileged mode. 10, 55

user application is an application designed to be executed at user-mode on a system managed by an operating system. 10

user-mode on RISC-V, the user-mode is equivalent to the User/Application mode. 10, 13, 128

Acronyms

ABI Application Binary Interface. 10, 54, 84, 91–94, 99, 102, 103, 125

ASCII American Standard Code for Information Interchange. 10, 16–18

bit Binary digit. 2–5, 10–20, 22, 23, 25, 28, 29, 35, 37, 40, 44–46, 48–50, 102, 108–114, 120, 121, 125

CPU Central Processing Unit. 2–5, 10–13, 32, 36, 49, 52, 107–112, 114–125, 128–132

CSR Control and Status Register. 10, 120–125, 129–132

ISA Instruction Set Architecture. 4, 10, 13, 24–26, 38–40, 49, 50, 55, 109–111, 120, 124, 125, 128, 129

ISR Interrupt Service Routine. 10, 117–120, 124, 125

PC Program Counter. 4, 5, 10, 27, 32, 117, 121–124, 129, 131

UTF-8 Universal Coded Character Set (or Unicode) Transformation Format - 8-bit. 10, 16–18

Part I

Introduction to computer systems and assembly language

Chapter 1

Execution of programs: a 10,000 ft overview

There are several ways of encoding a computer program. Some programs, for example, are encoded using abstract instruction sets and are executed by emulators or virtual machines, which are other programs designed to interpret and execute the abstract instruction set. Bash scripts, Java byte-code programs, and Python scripts are common examples of programs that are encoded using abstract instruction sets and require an emulator or a virtual machine to support their execution.

A **native program is a program encoded using instructions that can be directly executed by the computer hardware, without help from an emulator or a virtual machine**. In this book, we focus our discussion on native programs. Hence, from now on, whenever we use the term "program", unless stated otherwise, we are referring to native programs.

Native program instructions usually perform simple operations, such as adding or comparing two numbers, nonetheless, by executing multiple instructions, a computer is capable of solving complex problems.

Most modern computers are built using digital electronic circuitry. These machines usually represent information using voltage levels that are mapped to two states, HIGH and LOW, or "1" (one) and "0" (zero). Hence, the basic unit of information on modern computers is a binary digit, *i.e.*, "1" or "0". Consequently, information and instructions are encoded as sequences of **bi**nary digi**t**s, or **bit**s.

1.1 Main components of computers

Computers are usually composed of the following main components:

- **Main memory**: The main memory is used to store the instructions and data of programs that are being executed. The main memory is usually volatile, hence, if the computer is turned off, its contents are lost.

- **Central Processing Unit**: the Central Processing Unit, or CPU, is the component responsible for executing the computer programs. The CPU retrieves programs' instructions from the main memory for execution. Also, when executing instructions, the CPU often reads/writes data from/to the main memory.

- **Persistent storage**: Since the main memory is volatile, there is usually a persistent storage device to preserve the programs and data when the power is shut down. Hard disk drives (HDDs), solid state drives (SSDs), and flash drives are example of persistent storage devices.

- **Peripherals**: Peripherals are input/output, or I/O, devices that are connected to the computer. Examples of peripheral devices include video cards (also known as graphics cards), USB controllers, network cards, *etc.*.

- **Bus**: The bus is a communication system that transfers information between the computer components. This system is usually composed of wires that are responsible for transmitting the information and associated circuitries, which orchestrate the communication.

Figure 1.1 illustrates a computer system in which the CPU, the main memory, a persistent storage device (HDD) and two I/O devices are connected through a system bus.

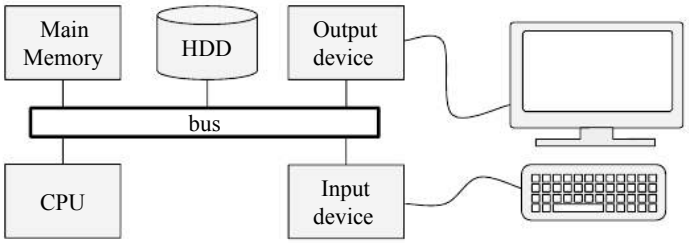


Figure 1.1: Computer system components connected through a system bus.

1.1.1 The main memory

The computer main memory is a storage device used to store the program instructions and data, and it is composed of a set of memory words. Each memory word is capable of storing a set of bits (usually eight bits) and is identified by a unique number, known as the memory word address. A byte addressable memory is a memory in which each memory word (a.k.a. memory location) stores a single byte and is associated with a unique address. Figure 1.2 illustrates the organization of a byte addressable memory. Notice that the memory word identified by address 5 (or simply "memory word 5") contains the value 11111111_2 while memory word 0 contains the value 00110110_2.

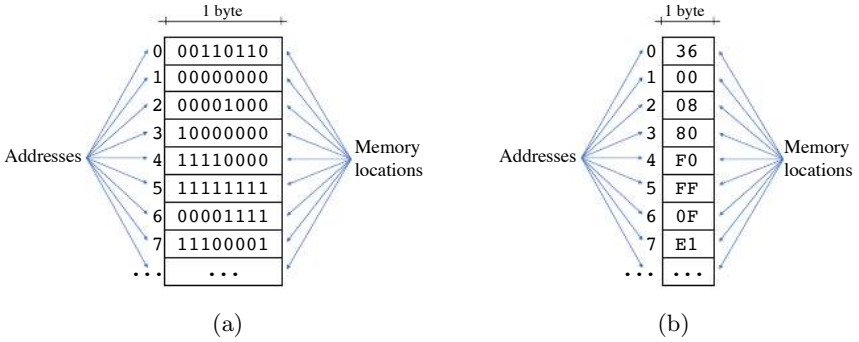


Figure 1.2: Organization of a byte addressable memory with its contents represented in the binary (a) and the hexadecimal (b) bases.

1.1.2 The CPU

The Central Processing Unit is the component responsible for executing the computer programs. There are several ways of implementing and organizing a CPU, however, to understand how programs are executed, it suffices to know that the CPU contains:

- **Registers**: a CPU register is a small memory device located inside the CPU. The CPU usually contains a small set of registers. RISC-V processors, for example, contain thirty-one 32-bit registers[1] that can be used by programs to store information inside the CPU. Computers often contain instructions that

[1] A 32-bit register is a register that is capable of storing 32 bits, *i.e.*, values composed of 32 bits.

copy values from the main memory into CPU registers, known as "load" instructions, and instructions that copy values from the CPU registers into the main memory, known as "store" instructions.

- **A datapath**: the CPU datapath is responsible for performing operations, such as arithmetic and logic operations, on data. The datapath usually performs the operation using data from the CPU registers and store the results on CPU registers.

- **A control unit**: the control unit is the unit responsible for orchestrating the computer operation. It is capable of controlling the datapath and other components, such as the main memory, by sending commands through the bus. For example, it may send a sequence of commands to the datapath and to the main memory to orchestrate the execution of a program instruction.

Accessing data on registers is much faster than accessing data on the main memory. Hence, programs tend to copy data from memory and keep them on CPU registers to enable faster processing. Once the data is no longer needed, it may be discarded or saved back on the main memory to free CPU registers.

The Instruction Set Architecture, or ISA, defines the computer instructions set, including, but not limited to, the behavior of the instructions, their encoding, and resources that may be accessed by the instructions, such as CPU registers. A program that was generated for a given ISA can be executed by any computer that implements a compatible ISA.

ISAs tend to evolve over time, however, ISA designers try to keep newer ISA versions compatible with previous ones so that legacy code, *i.e.*, code generated for previous versions of the ISA, can still be executed by newer CPUs. For example, a program that was generated for the 80386 ISA can be executed by any processor that implements this or any other compatible ISAs, such as the 80486 ISA.

1.2 Executing program instructions

As discussed previously, modern computers usually store the program that is being executed on main memory, including its instructions and data. The CPU retrieves programs' instructions from the main memory for execution. Also, when executing instructions, the CPU may read/write data from/to the main memory. To illustrate this process we will consider a CPU that implements the RV32I ISA.

The RV32I ISA specifies that instructions are encoded using 32 bits. Hence, assuming the system has a byte addressable memory[2], each instruction occupies four memory words. Also, it specifies that instructions are executed sequentially[3], in the same order they appear in the main memory.

Let us consider a small program generated for the RV32I ISA that is composed of three instructions and is stored in main memory starting at address 8000. Since each instruction occupies four bytes (*i.e.*, 32 bits) and instructions are stored consecutively on main memory, the first instruction is located at addresses 8000, 8001, 8002, and 8003, the second one on addresses 8004, 8005, 8006, and 8007, and the third one on addresses 8008, 8009, 800A, and 800B. Figure 1.3 illustrates the instructions stored on the main memory.

The CPU usually contains a register to keep track of the next instruction that needs to be executed. This register, called Program Counter, or PC, on the RV32 ISA, stores the starting address of the sequence of memory words that store the next instruction to be executed. For example, before executing the first instruction of the code illustrated at Figure 1.3, the PC contains the value 8000. Once the instruction stored at address 8000 is fetched, the value of the PC is updated by adding four to its contents so that the next instruction (at address 8004) can be fetched for execution once the current instruction is completed. Algorithm 1 illustrates the execution cycle

[2]This is usually the case in modern computers.
[3]As discussed in Section 6.7, control-flow instructions may change the normal execution flow.

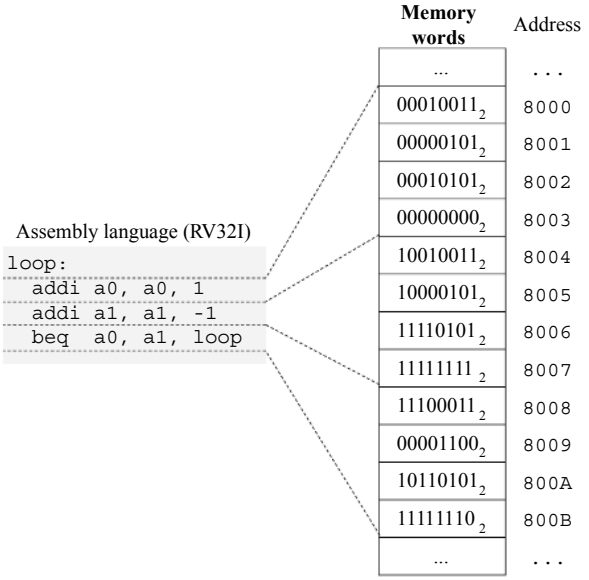


Figure 1.3: Three RV32I instructions stored on a byte addressable memory starting at address 8000.

performed by a simple RV32I CPU. First, the CPU uses the address in the PC to fetch an instruction (a sequence of four memory words, *i.e.*, 32 bits) from main memory and store it on an internal register called `IR`. Then, it updates the PC so it points to the next instruction in memory. Finally, it executes the instruction that was fetched from memory. Notice that when executing the instruction, the CPU may also access the main memory to retrieve or store data.

Algorithm 1: RV32I instructions execution cycle.

```
1 while True do
2 |   // Fetch instruction and update PC ;
3 |   IR ← MainMemory[PC] ;
4 |   PC ← PC+4;
5 |   ExecuteInstruction(IR);
6 end
```

To execute a program, the operating system essentially loads the program into the main memory (*e.g.*, from a persistent storage device) and sets the PC so it points to the program entry point.

1.3 The boot process

Since the main memory is volatile, whenever a computer is powered on, it contains garbage. As a consequence, at this point, the CPU may not retrieve instructions from the main memory. In this context, on power on, the PC is automatically set so that the CPU starts by retrieving instructions from a small non-volatile memory device, which stores a small program that performs the boot process[4]. This program sets up the basic computer components, checks the boot configuration (also stored on a non-volatile memory), and, based on its settings, loads into main memory the operating system boot loader from a persistent storage device (*e.g.*, the hard disk drive).

Once the operating system boot loader is loaded into memory, the CPU starts executing its code, which, in turn, finishes setting up the computer and loading the primary operating system modules into the main memory. Once the boot process finishes, a copy of the primary operating system modules is located in the main memory, and the system is ready to execute other programs, such as users' programs.

[4]In old personal computer systems this program is known as the Basic Input/Output System, or BIOS. More modern computers use the Unified Extensible Firmware Interface, or UEFI, standard.

Chapter 2

Data representation on modern computers

This chapter discusses how information is represented on computers. First, Section 2.1 introduces the concepts of numeral systems and the positional notation. Then, sections 2.2 and 2.3 discuss how numbers and text are represented on computers, respectively. Next, Section 2.4 shows how data is organized in memory. Finally, Section 2.5 discusses how instructions are encoded.

2.1 Numeral Systems and the Positional Notation

A **numeral system is a system used for expressing numbers**. The numeral system defines a set of symbols and rules for using them to represent a given set of numbers (*e.g.*, the natural numbers). For example, the "unary numeral system", is a numeral system in which every natural number is represented by a corresponding number of symbols. In this system, assuming the base symbol is the $\star$ character, natural numbers one, two, and five would be represented as $\star$, $\star\star$, and $\star\star\star\star\star$, respectively.

The "decimal numeral system" is the most common numeral system used by humans to represent integer and non-integer numbers. Let:

- D_{10} be the set of symbols used in the decimal numeral system, *i.e.*, $D_{10} = \{$"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"$\}$; and
- d^i be a digit on a number represented in the decimal numeral system, *i.e.*, $d^i \in D_{10}$;
- $number_{10}$ be a number represented in the decimal numeral system;

any natural number may be represented on the decimal numeral system by an ordered sequence of m digits, as illustrated by Equation 2.1. The superscript annotation indicates the digit position. Notice that the rightmost digit is associated with position 0 and the leftmost digit is associated with position $m - 1$.

$$number_{10} = d^{m-1}d^{m-2}\cdots d^1 d^0_{10} \tag{2.1}$$

For example, the number one thousand nine hundred and sixty nine is represented by the sequence 1969. In this case, $m = 4$, $d^3 =$ "1", $d^2 =$ "9", $d^1 =$ "6", and $d^0 =$ "9"

Let:

- $symbol\_value(d^i)$ be the value of the symbol used in digit d^i; and
- $symbol\_value($"0"$) =$ zero, $symbol\_value($"1"$) =$ one, $symbol\_value($"2"$) =$ two, ..., $symbol\_value($"9"$) =$ nine;

The value of a number with m digits in the decimal numeral system is computed by Equation 2.2.

$$value(number_{10}) = \sum_{i=0}^{i<m} \underbrace{symbol\_value(d^i) \times 10^i}_{\text{digit value}} \tag{2.2}$$

Notice that the value of each digit d^i, *i.e.*, the contribution of each digit d^i to the number value, depends on the symbol used and its position (index i) on the sequence. For example, the number 1969 has four digits: $d^3 = 1, d^2 = 9, d^1 = 6, d^0 = 9$. The value of digit d^2 is nine hundred while the value of digit d^0 is nine.

A positional numeral system is a numeral system in which the value of a digit d^i depends on the value of the symbol used on the digit and also on the position of the digit on the sequence of digits. The decimal numeral system is a positional numeral system. The unary and the Roman numeral systems, on the other hand, are common examples of non-positional numeral systems.

The base, or radix, of a numeral system is the number of different symbols a digit may assume to represent the numbers. The base of the "unary numeral system" is one while the base of the "decimal system" is ten.

The decimal, binary, octal and hexadecimal numeral systems are positional numeral systems that are frequently used when programming computers. The only difference between these numeral systems is the base. While the base of the decimal numeral system is ten, the base of the binary, the octal, and the hexadecimal numeral systems are two, eight, and sixteen, respectively.

Let:

- $base$ be the base, or radix, of the positional numeral system;
- D_{base} be the set of symbols used in the positional numeral system (*e.g.*, $D_2 = \{$"0", "1"$\}$); and
- d^i_{base} be the i^{th} digit on a number represented in the positional numeral system, *i.e.*, $d^i_{base} \in D_{base}$;
- $number_{base}$ be a number represented in the numeral system;

any natural number may be represented on the decimal/binary/octal/hexadecimal positional numeral system by an ordered sequence of m digits, as illustrated by Equation 2.3.

$$number_{base} = d^{m-1}_{base} d^{m-2}_{base} \cdots d^1_{base} d^0_{base} \tag{2.3}$$

Let:

- $symbol\_value(d^i_{base})$ be the value of the symbol used in digit d^i_{base}; and
- $symbol\_value($"0"$) =$ zero, $symbol\_value($"1"$) =$ one, and so on;

The value of a natural number with m digits in any of these positional numeral systems is defined by Equation 2.4.

$$value(number_{base}) = \sum_{i=0}^{i<m} \underbrace{symbol\_value(d^i_{base}) \times base^i}_{\text{digit value}} \tag{2.4}$$

For example, the value of any natural number represented on the binary numeral system ($base = 2$) is defined by Equation 2.5

$$value(number_2) = \sum_{i=0}^{i<m} \underbrace{symbol\_value(d^i_2) \times 2^i}_{\text{digit value}} \tag{2.5}$$

Notice that the value of the sequence 11 is three on the binary numeral system ($1 \times 2^1 + 1 \times 2^0$) while it is eleven on the decimal numeral system ($1 \times 10^1 + 1 \times 10^0$) and seventeen on the hexadecimal numeral system ($1 \times 16^1 + 1 \times 16^0$).

When working with multiple numeral systems it is often necessary to annotate the numbers so that it is possible to identify the numeral system being used, and hence, its value. A common notation is to append a subscribed suffix to the number indicating the base of the positional numeral system. For example, the value of number 11_{10} is eleven while the value of number 11_2 is three[1].

Appending a subscribed suffix to the number is not a natural way of annotating numbers in computer programs. In these cases, a common approach is to append a prefix that indicates the base. For example, in "C", the programmer may use the prefix "`0b`"/"`0`"/"`0x`" to indicate that the number is in the binary/octal/hexadecimal base, *i.e.*, base 2/8/16. In "C", numbers that lack a prefix belong to the decimal numeral system.

The binary and octal numeral systems use a subset of the symbols used on the decimal numeral system to represent the numbers. The hexadecimal numeral system, on the other hand, requires more than ten symbols, hence, new symbols are needed. In this case, the first letters of the alphabet are used to complement the set of symbols. Table 2.1 shows the symbols used in each one of these positional numeral systems and their corresponding values.

Symbol	*symbol_value*	Used in base			
		2	8	10	16
"0"	zero	✓	✓	✓	✓
"1"	one	✓	✓	✓	✓
"2"	two		✓	✓	✓
"3"	three		✓	✓	✓
"4"	four		✓	✓	✓
"5"	five		✓	✓	✓
"6"	six		✓	✓	✓
"7"	seven		✓	✓	✓
"8"	eight			✓	✓
"9"	nine			✓	✓
"A"	ten				✓
"B"	eleven				✓
"C"	twelve				✓
"D"	thirteen				✓
"E"	fourteen				✓
"F"	fifteen				✓

Table 2.1: Set of symbols used in binary, octal, decimal, and hexadecimal numeral systems and their respective values.

Table 2.2 shows how values zero to twenty are represented in the hexadecimal, decimal, octal, and binary numeral systems.

2.1.1 Converting numbers between bases

Converting numbers between positional numeral systems is a common task in several contexts, specially when programming or debugging a computer system. Since humans usually prefer to think using the decimal numeral system, they usually convert numbers from other positional numeral systems to the decimal numeral system to reason about their values. Also, in some situations, it may be necessary to convert values to the binary, octal, or the hexadecimal numeral systems.

Equation 2.4 shows how to compute the value of numbers represented on any positional numeral system.

To represent a value V in a positional numeral system, one must find a sequence of digits $d^{m-1}_{base} d^{m-2}_{base} \cdots d^{1}_{base} d^{0}_{base}$ so that Equation 2.4 holds. In other words, let $v(d^i_b)$ be the value of the symbol used on the i^{th} digit[2] of a number represented in base b,

[1] Notice that, accordingly to Equation 2.4, $value(11_2) = 1 \times 2^1 + 1 \times 2^0 = \text{three}$

[2] We will use the notation $v(d^i_b)$ instead of $symbol\_value(d^i_b)$ to keep equations short.

Value	Numeral system			
	Hexadecimal	Decimal	Octal	Binary
zero	0_{16}	0_{10}	0_8	0_2
one	1_{16}	1_{10}	1_8	1_2
two	2_{16}	2_{10}	2_8	10_2
three	3_{16}	3_{10}	3_8	11_2
four	4_{16}	4_{10}	4_8	100_2
five	5_{16}	5_{10}	5_8	101_2
six	6_{16}	6_{10}	6_8	110_2
seven	7_{16}	7_{10}	7_8	111_2
eight	8_{16}	8_{10}	10_8	1000_2
nine	9_{16}	9_{10}	11_8	1001_2
ten	A_{16}	10_{10}	12_8	1010_2
eleven	B_{16}	11_{10}	13_8	1011_2
twelve	C_{16}	12_{10}	14_8	1100_2
thirteen	D_{16}	13_{10}	15_8	1101_2
fourteen	E_{16}	14_{10}	16_8	1110_2
fifteen	F_{16}	15_{10}	17_8	1111_2
sixteen	10_{16}	16_{10}	20_8	10000_2
seventeen	11_{16}	17_{10}	21_8	10001_2
eighteen	12_{16}	18_{10}	22_8	10010_2
nineteen	13_{16}	19_{10}	23_8	10011_2
twenty	14_{16}	20_{10}	24_8	10100_2

Table 2.2: Values zero to twenty represented in the hexadecimal, decimal, octal, and binary numeral systems.

one must find a sequence of digits $d_{base}^{m-1} d_{base}^{m-2} \cdots d_{base}^{1} d_{base}^{0}$ so that:

$$V = v(d_b^{m-1}) \times b^{m-1} + \cdots + v(d_b^1) \times b^1 + v(d_b^0) \times b^0 \tag{2.6}$$

The previous equation may be rewritten as:

$$V = b \times (\underbrace{v(d_b^{m-1}) \times b^{m-2} + \cdots + v(d_b^1) \times b^0}_{V'}) + v(d_b^0) \tag{2.7}$$

Notice that $v(d_b^0)$ and V' are equivalent to the remainder and the quotient of the division of V by b. Hence, to find out the symbol value of digit d_b^0 (*i.e.*, $v(d_b^0)$) it suffices to compute the remainder of the division of V by b.

Using the same reasoning, the symbol value of digit d_b^1 may be computed by dividing V' by b. Notice that the remainder of the division of V' by b is equal to $v(d_b^1)$.

Let $symbol\_from\_value(v, b)$ be a function that returns the symbol used to represent value v on base b (*e.g.*, $symbol\_from\_value(eleven, 16) =$ "B"), Algorithm 2 shows an algorithm to compute the sequence of digits $d_b^{m-1} d_b^{m-2} \cdots d_b^1 d_b^0$ that repre-

sent value V on base b.

Algorithm 2: Algorithm to compute the sequence of digits $d_b^{m-1} d_b^{m-2} \cdots d_b^1 d_b^0$ that represent value V on base b.

input : Value V and base b.
output: Sequence of digits $d_b^{m-1} d_b^{m-2} \cdots d_b^1 d_b^0$.

```
1 i = 0 ;
2 tmp = V ;
3 while tmp ≠ 0 do
4 |   rem = tmp mod b ;
5 |   d_b^i = symbol_from_value(rem, b) ;
6 |   tmp = tmp / b ;
7 |   i = i + 1 ;
8 end
```

To illustrate the use of Algorithm 2 lets compute the sequence of digits d_2^{m-1} d_2^{m-2} $\cdots$ d_2^1 d_2^0 that represent value twenty six on base 2. First, we divide twenty six by two. The remainder of this division is zero and the quotient is thirteen, hence, $v(d_2^0)$ = zero and d_2^0 = "0". This process is illustrated in Figure 2.1 (a). To compute $v(d_2^1)$ we now divide thirteen (*i.e.*, the quotient of the last division) by two. The remainder of this division is one and the quotient is six, hence, $v(d_2^1)$ = one and d_2^1 = "1". This partial result is illustrated in Figure 2.1 (b). To compute $v(d_2^2)$ we now divide six (the previous quotient) by two. The remainder of this division is zero and the quotient is three, hence, $v(d_2^2)$ = zero and d_2^2 = "0". Now, we divide three by two. The remainder of this division is one and the quotient is one, hence, $v(d_2^3)$ = one and d_2^3 = "1". Again, we divide the last quotient (one) by two. The remainder of this division is one and the quotient is zero, hence, $v(d_2^4)$ = one and d_2^4 = "1". Since the last quotient is zero, the process is complete. As a result, the sequence 11010_2 represents value twenty six on the binary numeral system. Figure 2.1 (c) shows all the quotients and remainders computed during the execution of Algorithm 2.

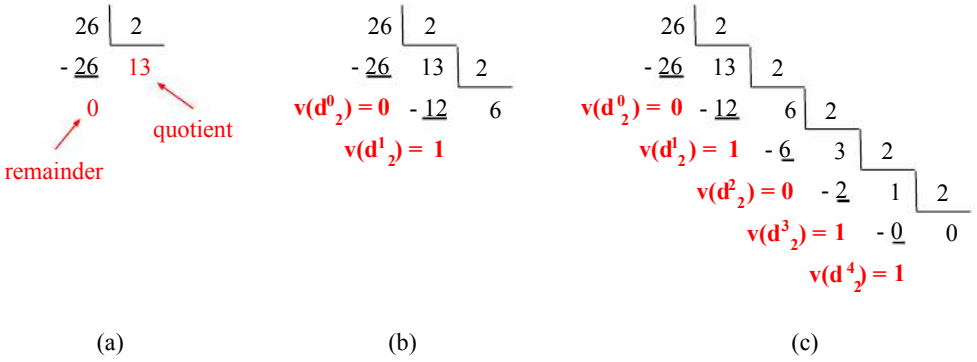


Figure 2.1: Example of applying Algorithm 2 to compute the sequence of digits $d_2^{m-1} d_2^{m-2} \cdots d_2^1 d_2^0$ that represent value twenty six on base 2. (a) First iteration of the algorithm. (b) Values of $v(d_2^0)$ and $v(d_2^1)$ computed after two iterations. (c) Final result: 11010_2 = twenty six.

Figure 2.2 shows an example of applying Algorithm 2 to compute the sequence of digits $d_{16}^{m-1} d_{16}^{m-2} \cdots d_{16}^1 d_{16}^0$ that represent value twenty six on base 16.

To convert a number from a positional numeral system with base A to a number on a positional numeral system with base B, one may use Equation 2.4 to compute the value of the number in base A and Algorithm 2 to compute the sequence of digits $d_B^{m-1} d_B^{m-2} \cdots d_B^1 d_B^0$ that represent the computed value on base B.

Representing numbers in the hexadecimal numeral system is more compact than representing numbers in the binary numeral system. For example, the representation of number one thousand in hexadecimal ($3E8_{16}$) requires only three digits while in binary (1111101000_2) it requires ten digits. Also, since both bases are power of two,

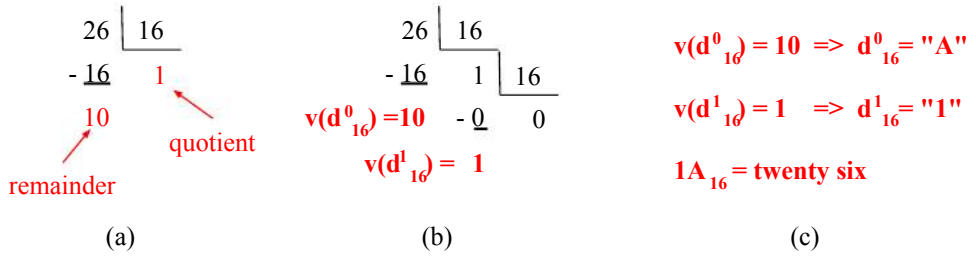


Figure 2.2: Example of applying Algorithm 2 to compute the sequence of digits $d_{16}^{m-1} d_{16}^{m-2} \cdots d_{16}^{1} d_{16}^{0}$ that represent value twenty six on base 16. (a) First iteration of the algorithm. (b) Values of $v(d_{16}^{0})$ and $v(d_{16}^{1})$ computed after two iterations. (c) Final result: $1A_{16}$ = twenty six.

converting between these bases can be done by replacing subsets of consecutive four bits by single hexadecimal digits, and vice versa. For example, Equation 2.8 illustrates how number $3E8_{16}$ can be converted to binary and Equation 2.9 shows how number 101101011101012 can be converted to hexadecimal.

$$\underbrace{3}_{0011_2}\ \underbrace{E}_{1110_2}\ \underbrace{8}_{1000_2}{}_{16} = 001111101000_2 = 1111101000_2 \tag{2.8}$$

$$\underbrace{10}_{2_{16}}\ \underbrace{1101}_{D_{16}}\ \underbrace{01110101}_{7_{16}\ 5_{16}}{}_{2} = 2D75_{16} \tag{2.9}$$

2.2 Representing numbers on computers

A **binary digit**, or **bit**, is a digit in the binary numeral system, *i.e.*, a digit that may assume one of two values, "0" (zero) or "1" (one).

Most modern computers are built using digital electronic circuitry. These machines usually represent information using voltage levels that are mapped to two states, HIGH and LOW, or "1" (one) and "0" (zero). Hence, the basic unit of information on modern computers is a bit. Consequently, numbers on computers are represented by a sequence of binary digits, or bits.

2.2.1 Unsigned numbers

On computers, unsigned numbers are represented using the binary positional numeral system with m bits. In other words, unsigned numbers are represented by a sequence of m binary digits (or bits) $d_2^{m-1}\ d_2^{m-2} \cdots d_2^1\ d_2^0$ and the value of the number is defined by Equation 2.5.

Unsigned numbers are a subset of the natural numbers ($\mathbb{N}$). The natural numbers set has infinite numbers (zero to infinite), however, the set of unsigned numbers that can be represented on computers depends on the amount of bits being used to represent the number. For example, if only 8 bits are being used to represent the number, then, only 256 numbers can be represented. In this case, the smallest number is zero (00000000_2) and the largest one is 255 (11111111_2). The unsigned numbers' representation can represent natural numbers in the range $[0\ ..\ (2^m - 1)]$ with a sequence of m bits.

Typically, programming language types are mapped by the compiler to a native type, *i.e.*, a type known by the computer architecture. For example, on a 32-bit computer, the "C" `unsigned integer` type is mapped by the compiler to a "32-bit word". In other words, on a 32-bit computer, an `unsigned integer` can represent numbers from zero to $2^{32} - 1$. In this context, zero is represented by:

$$000000000\,000000000\,000000000\,000000000_2 \tag{2.10}$$

while $2^{32} - 1$ is represented by

$$11111111\,11111111\,11111111\,11111111_2 \tag{2.11}$$

To illustrate this concept, Table 2.3 shows the list of unsigned numbers that can be represented with three-bit words.

Three-bit word	Value in the unsigned representation
000	0_{10}
001	1_{10}
010	2_{10}
011	3_{10}
100	4_{10}
101	5_{10}
110	6_{10}
111	7_{10}

Table 2.3: Unsigned numbers that can be represented with three-bit words.

2.2.2 Signed numbers

On computers, signed numbers are a subset of the integer numbers ($\mathbb{I}$). The set of signed numbers usually include negative and non-negative numbers. While the integer numbers set is infinite, the set of signed numbers that can be represented on computers depends on the amount of bits being used to represent the number and on the signed number representation. The most common signed number representation is the "two's complement". The next sections presents the "signal and magnitude", the "one's complement", and the "two's complement" representation.

Signal and magnitude

The "signal and magnitude" is a number representation that can be used to represent signed numbers. In this representation, signed numbers are represented as a sequence of m bits so that bit d_2^{m-1} represents the signal of the number and the remaining bits, *i.e.*, bits d_2^{m-2} to d_2^0, represent the magnitude. The magnitude value can be computed using Equation 2.5. In this context, in case bit d_2^{m-1} is "1", then the number is negative, otherwise, it is non-negative. Table 2.4 show the values of three-bit words on the unsigned and signal and magnitude representations.

Three-bit word	Value	
	unsigned	signal and magnitude
000	0_{10}	0_{10}
001	1_{10}	1_{10}
010	2_{10}	2_{10}
011	3_{10}	3_{10}
100	4_{10}	-0_{10}
101	5_{10}	-1_{10}
110	6_{10}	-2_{10}
111	7_{10}	-3_{10}

Table 2.4: Values of three-bit words on the unsigned and signal and magnitude representations.

The "signal and magnitude" representation can represent numbers in the range $[-(2^{m-1} - 1) \,..\, (2^{m-1} - 1)]$ with a sequence of m bits.

One's complement

The "one's complement" representation is a number representation that can be used to represent signed numbers. In this representation, signed numbers are represented as a sequence of m bits so that bit d_2^{m-1} represents the signal of the number. If bit d_2^{m-1} is "1", then the number is negative, otherwise, it is non-negative.

The magnitude of a non-negative number represented in one's complement (*i.e.*, a number in which d_2^{m-1} = "0") is computed in the same way the value of unsigned numbers are computed, *i.e.*, using Equation 2.5. For example, the three-bit number "010" is a non-negative number in one's complement representation and its magnitude is two, since $0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = two$.

The magnitude of a negative number represented in one's complement (*i.e.*, a number in which d_2^{m-1} = "1") is computed by first "complementing" (inverting) all bits and then using Equation 2.5. For example, the three-bit number "110" is a negative number in one's complement representation and its magnitude is one, since the value of its complement ("001") is one ($0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = one$).

Table 2.5 shows the values of three-bit words on the unsigned, signal and magnitude, and one's complement representations. Similar to the signal and magnitude representation, there are two representations for value zero on the one's complement representation.

Three-bit word	Value			
	unsigned	signal and magnitude	one's complement	two's complement
000	0_{10}	0_{10}	0_{10}	0_{10}
001	1_{10}	1_{10}	1_{10}	1_{10}
010	2_{10}	2_{10}	2_{10}	2_{10}
011	3_{10}	3_{10}	3_{10}	3_{10}
100	4_{10}	-0_{10}	-3_{10}	-4_{10}
101	5_{10}	-1_{10}	-2_{10}	-3_{10}
110	6_{10}	-2_{10}	-1_{10}	-2_{10}
111	7_{10}	-3_{10}	-0_{10}	-1_{10}

Table 2.5: Values of three-bit words on the unsigned, signal and magnitude, one's complement and two's complement representations.

The "one's complement" representation can represent numbers in the range $[-(2^{m-1}-1) \,..\, (2^{m-1}-1)]$ with a sequence of m bits.

Two's complement

The "two's complement" representation is also a number representation that can be used to represent signed numbers. Also, in this representation, signed numbers are represented as a sequence of m bits so that bit d_2^{m-1} represents the signal of the number. If bit d_2^{m-1} is "1", then the number is negative, otherwise, it is non-negative.

The magnitude of a non-negative number represented in two's complement (*i.e.*, a number in which d_2^{m-1} = "0") is computed in the same way the value of unsigned numbers are computed, *i.e.*, using Equation 2.5. For example, the three-bit number "010" is a non-negative number in two's complement representation and its magnitude is *two*, since $0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = two$.

The magnitude of a negative number represented in two's complement (*i.e.*, a number in which d_2^{m-1} = "1") is computed by first "complementing" (inverting) all bits, then adding one, and, finally, using Equation 2.5. For example, the three-bit number "110" is a negative number in two's complement representation and its magnitude is *two*, since the value of its complement ("001") plus one is two.

Table 2.5 shows the values of three-bit words on the unsigned, signal and magnitude, one's complement, and two's complement representations. Notice that there is only one representation for value zero on the two's complement representation.

Another interesting characteristic of the two's complement representation is that the addition, subtraction, and multiplication of two m-bit numbers in this representation are identical to the addition, subtraction, and multiplication operations of two m-bit numbers in the unsigned binary representation[3]. As a consequence, the same hardware may be used to perform these operations both on unsigned numbers and signed numbers represented in "two's complement". Figure 2.3 shows an example in which two numbers are added and the same sequence of bits is produced both in the unsigned numbers representation and the two's complement representation.

Sequence of bits	Unsigned representation	Two's complement representation
010_2	2_{10}	2_{10}
$+\underline{100_2}$	$+\underline{4_{10}}$	$+\underline{-4_{10}}$
110_2	6_{10}	-2_{10}

Figure 2.3: The addition of two numbers produces the same sequence of bits both in the unsigned numbers representation and the two's complement representation.

The two's complement representation can represent numbers in the range $[-(2^{m-1}) \,..\, (2^{m-1}-1)]$ with a sequence of m bits.

NOTE: The two's complement is the most common method to represent signed integer numbers on modern computers.

2.2.3 Binary arithmetic and Overflow

Arithmetic on unsigned binary numbers is similar to arithmetic on unsigned decimal numbers. When adding two unsigned binary numbers[4], digits are added one by one, from the least significant one (d_2^0) to the most significant one (d_2^{m-1}), *i.e.*, from the rightmost digit to the leftmost digit. In some cases, the addition of two digits produces a value that cannot be represented by a single digit. In these cases, a carry out is produced and its value must be added to the next most significant digit. In the binary representation, this situation occurs when the result is greater than one.

Figure 2.4 (a) illustrates the addition of two three-bit unsigned binary numbers. First, the two least significant (rightmost) digits are added together. Since the result (two=10_2) cannot be represented by a single binary digit, the least significant digit of the result, *i.e.*, zero, is placed in the current position and a carry is produced and placed on top of the next digit (the red arrow illustrates this process). Then, the second least significant digits (1 and 1) must be added. In this case, since there is a carry from the previous digit addition, the carry is also added to both digits. The result of this addition is three (11), hence, another carry is produced and placed on top of the most significant digit. Again, the least significant digit of the result, *i.e.*, one, is placed in the current position. Finally, the most significant digits are added together with their carry and the result is one. Figure 2.4 (b) illustrates the same addition using a simplified (cleaner) representation.

Subtraction of unsigned binary numbers is also performed in a similar way decimal numbers are subtracted. When subtracting two unsigned binary numbers, digits are subtracted one by one, from the least significant one (d_2^0) to the most significant one (d_2^{m-1}), *i.e.*, from the rightmost digit to the leftmost digit. In some cases, the subtraction is not possible because the first operand is smaller than the second one.

[3]This is only true if the output is represented using the same number of bits as the inputs (*i.e.*, m bits) and if any overflow beyond these bits are discarded. This is usually the case in modern computers.

[4]As discussed in previous section, adding (subtracting) two m-bit unsigned binary numbers produces the same sequence of m-bits as adding (subtracting) two m-bit signed numbers on the two's complement representation. Hence, the same approach used for unsigned binary numbers may be used to perform addition and subtraction operations on signed numbers using the two's complement representation.

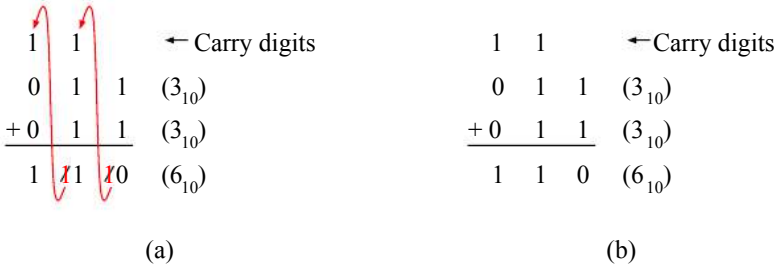


Figure 2.4: Adding two three-bit binary numbers. (a) Red arrows indicate where the carry out comes from. (b) Simplified representation (without arrows).

In these cases, "some value" is borrowed from the left digit and this borrowed value must be accounted for when performing the subtraction on the left digit.

Figure 2.5 illustrates the subtraction of two three-bit unsigned binary numbers. Figure 2.5 (b) shows the first step, in which the least significant digits are subtracted. Since "1" cannot be subtracted from "0", some value must be borrowed from the left column. The "*" character indicates that value had to be borrowed from the left column. The result in this column is "1", since "10" (two) minus "1" (one) is "1". Figure 2.5 (c) illustrates the operation on the second least significant digit. Since the right column borrowed from this column, the first operand is now "0". Again, since "1" cannot be subtracted from "0", some value must be borrowed from the left column. The result in this column is "1", since "10" (two) minus "1" (one) is "1". Figure 2.5 (d) shows the subtraction of the most significant digits (zero minus zero) and the final result. Figure 2.5 (e) shows a simplified representation of the subtraction.

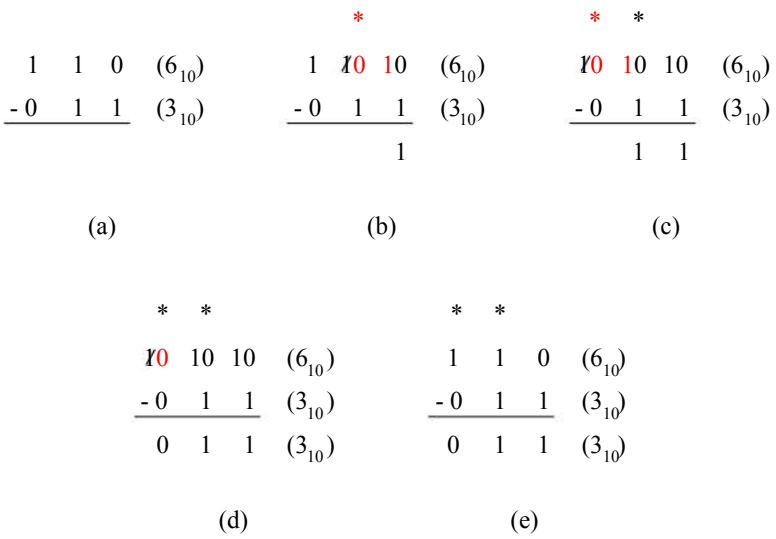


Figure 2.5: Subtraction of two three-bit binary numbers. (a) The digits of both numbers are aligned on columns. (b) First, the least significant digits are subtracted - the "*" character indicates that some value was borrowed from the left column. (c) The second least significant digits are subtracted - again, the "*" character indicates that some value was borrowed from the left column. (d) Finally, the most significant digits are subtracted producing digit "0". (e) Simplified representation of the subtraction.

2.2.4 Integer Overflow

An integer overflow occurs when the result of an arithmetic operation on two integer m-bit binary numbers is outside of the range that can be represented by an m-bit binary number. Figure 2.6 shows an example in which the addition of two three-bit

unsigned binary numbers causes an integer overflow. In this case, adding one to seven should result in eight, however, the value eight cannot be represented using only three bits on the unsigned binary representation.

$$\begin{array}{cccc} 1 & 1 & 1 & \\ & 0 & 0 & 1 \\ + & 1 & 1 & 1 \\ \hline & 0 & 0 & 0 \end{array} \quad \begin{array}{l} \leftarrow \text{carry digits} \\ (1_{10}) \\ (7_{10}) \\ (0_{10}) \end{array}$$

Figure 2.6: Example of an integer overflow on the unsigned binary representation. The result of one plus seven cannot be represented by a three-bit unsigned binary number.

Even though the operation illustrated on Figure 2.6 characterizes an integer overflow on the unsigned binary representation, it does not characterize an integer overflow on the signed (two's complement) binary representation. In this case, the operation is adding one (001) to minus one[5] (111) and the expected result, *i.e.*, zero, can be represented by a three-bit unsigned binary number.

Figure 2.7 shows an example in which the addition of two three-bit signed binary numbers using the two's complement method causes an integer overflow. In this case, however, there was not integer overflow on the unsigned binary number representation. Notice that the result of the operation is as expected, *i.e.*, four (100), on the unsigned binary representation.

$$\begin{array}{cccc} & 1 & 1 & \\ & 0 & 1 & 1 \\ + & 0 & 0 & 1 \\ \hline & 1 & 0 & 0 \end{array} \quad \begin{array}{l} \leftarrow \text{carry digits} \\ (3_{10}) \\ (1_{10}) \\ (-4_{10}) \end{array}$$

Figure 2.7: Example of an integer overflow on the signed binary representation. The result of three plus one cannot be represented by a three-bit signed binary number using the two's complement representation.

2.3 Representing text

A character is the basic unit of information when representing text on computers and usually corresponds to a letter (*e.g.*, "a"), a decimal digit (*e.g.*, "2"), a punctuation mark (*e.g.*, "." or "?"), white spaces, or even a control information[6].

The character encoding standard defines how characters are represented on computers. For example, the **American Standard Code for Information Interchange**, or **ASCII**, defines that characters are represented by seven-bit numbers. Table 2.6 shows a subset of the characters encoded by the American Standard Code for Information Interchange. Notice that the letter "a" is encoded as the number 97_{10} (1100001_2) while digit "2" is encoded as the number 50_{10} (0110010_2).

The ASCII character encoding standard was designed in the 1960s and, even though it included most symbols used on the English language, it lacked several important symbols required by other languages, such letters with accents (*e.g.*, "á", "ç", ...). In this context, several other character encoding standards were introduced, including an extension to the ASCII standard, the "Extended ASCII", or EASCII. Accordingly to Google[7], in 2008 the UTF-8 character encoding standard became the most common encoding for HTML files. As of 2020, a survey performed by the

[5] Notice that the three-bit sequence 111 represents the value minus one on the two's complement representation.

[6] Control characters are not intended to represent printable information. A line feed, carrier return and backspace are examples of control characters on computers.

[7] `https://googleblog.blogspot.com/2008/05/moving-to-unicode-51.html`

Binary	Hex.	Dec.	Char.
...			
0100001_2	21_{16}	33_{10}	!
0100010_2	22_{16}	34_{10}	”
...			
0101100_2	$2C_{16}$	44_{10}	,
0101101_2	$2D_{16}$	45_{10}	-
0101110_2	$2E_{16}$	46_{10}	.
0101111_2	$2F_{16}$	47_{10}	/
0110000_2	30_{16}	48_{10}	0
0110001_2	31_{16}	49_{10}	1
0110010_2	32_{16}	50_{10}	2
...			
0111000_2	38_{16}	56_{10}	8
0111001_2	39_{16}	57_{10}	9
...			

Binary	Hex.	Dec.	Char.
1000001_2	41_{16}	65_{10}	A
1000010_2	42_{16}	66_{10}	B
...			
1011001_2	59_{16}	89_{10}	Y
1011010_2	$5A_{16}$	90_{10}	Z
...			
1100001_2	61_{16}	97_{10}	a
1100010_2	62_{16}	98_{10}	b
...			
1111001_2	79_{16}	121_{10}	y
1111010_2	$7A_{16}$	122_{10}	z
...			
1111100_2	$7C_{16}$	124_{10}	|
1111101_2	$7D_{16}$	125_{10}	}
1111110_2	$7E_{16}$	126_{10}	~

Table 2.6: Subset of the characters encoded by the ASCII character encoding standard. Hex. and Dec. columns show the encoding value in hexadecimal and decimal representation while the Char. column shows the symbol encoded by the character.

W3Techs web site[8] indicated that more than 95.5 % of the world wide web websites are encoded with the UTF-8 character encoding standard.

The “Unicode (or Universal Coded Character Set) Transformation Format - 8-bit”, or UTF-8 for short, is a variable-width character encoding standard. In this standard, each character may be represented by one, two, three, or four bytes, *i.e.*, one, two, three, or four 8-bit numbers. Common characters, such as letters “a”, “b”, and “c”, are represented by a single byte, while more exotic ones are represented using multiple bytes. The euro currency sign (€), for example, is encoded using three bytes: 11100010_2, 10000010_2, and 10101100_2.

The UTF-8 standard was designed to be backward compatible with the ASCII standard. Hence, ASCII characters are represented on the UTF-8 standard using a single byte with the the same value. For example, letter “a” is represented by value ninety seven in both standards. In this way, a software designed to work with the UTF-8 standard can naturally open and handle ASCII encoded files.

Texts are represented in computers as sequences of characters on memory. For example, the word “Yes” is represented by a sequence of three characters (“Y”, “e”, and “s”) stored on consecutive memory positions. In case the ASCII character encoding standard is being used, the three consecutive memory positions will contain values 121_{10}, 101_{10}, and 115_{10}, respectively. Figure 2.8 illustrates how the word “maçã”[9] is represented in three different character encoding standards: the UTF-8, the ISO-LATIN-1 and the Mac OS Roman. Each square represents a byte and the values inside the squares are in hexadecimal. Notice that the UTF-8 standard requires two bytes to represent letter “ç” and two bytes to represent letter “ã”.

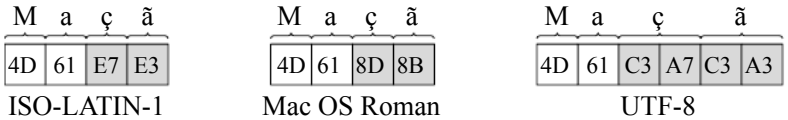


Figure 2.8: Word “maçã” represented in three different character encoding standards: the UTF-8, the ISO-LATIN-1 and the Mac OS Roman.

[8] `https://w3techs.com/technologies/overview/character_encoding`
[9] “Maçã” is the word for apple in Portuguese.

2.4 Organizing data on the memory

This section discusses how information is organized on the computer main memory. The discussion focus on byte addressable memories, which is the most common type of main memory used on modern computers.

2.4.1 Texts on the main memory

As discussed in Section 2.3, texts are represented in computers as sequences of characters on memory. The sequence is stored on consecutive memory words, *i.e.*, memory words with consecutive addresses.

ASCII characters are encoded using seven bits, however, when stored on a byte addressable memory, each character is usually stored in a single memory word, *i.e.*, they occupy eight bits[10]. Consequently, a five character text (*e.g.*, "hello") encoded with the ASCII standard is stored on five consecutive memory words.

UTF-8 characters are encoded using one, two, three, or four bytes, hence, when stored on a byte addressable memory, each character may require one, two, three, or four memory words. As illustrated by Figure 2.8, the UTF-8 character encoding standard requires six bytes to represent the word "maçã". Consequently, it requires six memory words to represent this word on a byte addressable memory.

In programming languages, the term "string" is often used to denote a sequence of characters. A NULL terminated string is a sequence of characters terminated by the character NULL, which is represented by value zero on most character encoding standards. A "C" string, *i.e.*, a string on the "C" programming language, is a NULL terminated string. Hence, in "C", the string "yes" takes four memory words, three to store the letters "y", "e", and "s", and another one to store the NULL character. The following program shows two different ways of declaring and initializing a string in a "C" program. The first approach (line 2), uses a sequence of symbols between quotes while the second one (line 3) uses an sequence of hexadecimal values using the array notation.

```
1  #include<stdio.h>
2  char name1[] = "John";
3  char name2[] = {0x4a, 0x6f, 0x68, 0x6e, 0x00};
4  int main()
5  {
6    printf("Name 1: \"%s\"\n", name1);
7    printf("Name 2: \"%s\"\n", name2);
8    printf("Size of name 1 = %d\n", sizeof(name1));
9    printf("Size of name 2 = %d\n", sizeof(name2));
10   return 0;
11 }
```

Both strings (`name1` and `name2`) in previous code require five memory words to be stored on memory[11]. In fact, since the hexadecimal values used in line 3 are the values for the "J", "o", "h", and "n" letters on the ASCII and UTF-8 encoding standards, both string are identical. The following listing shows the output of the previous program.

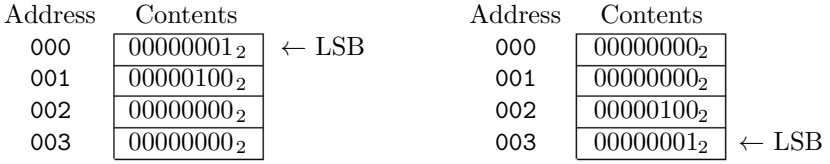
```
1 Name 1: "John"
2 Name 2: "John"
3 Size of name 1 = 5
4 Size of name 2 = 5
```

[10] In this case, the eighth bit is always set as zero.
[11] Notice that the string "John" is terminated by a NULL character in "C".

2.4.2 Numbers on the main memory

As discussed in Section 2.2, numbers on computers are represented by a sequence of m bits. In case m is greater than the memory word size, then, the sequence of m bits must be split and stored on multiple consecutive memory words. For example, a 32-bit number is split in four 8-bit parts and stored on four consecutive memory words on a byte addressable memory.

The endianness format refers to the order in which the bytes are stored on a computing system memory. There are two common formats: little-endian and big-endian. The little-endian format places the least significant byte (LSB) on the memory position associated with the lowest address while the big-endian format places the least significant byte on the memory position associated with the highest address. Figure 2.9 illustrates how the 32-bit number 00000000 00000000 00000100 00000001_2 (1025_{10}) can be stored on a byte addressable memory starting on address `000` in both formats. Notice that in the little-endian format (Figure 2.9 (a)), the least significant byte (00000001_2) is stored in address 000 while in the big-endian format (Figure 2.9 (b)), the least significant byte (00000001_2) is stored in address 003.



(a) little-endian (b) big-endian

Figure 2.9: 32-bit number 00000000 00000000 00000100 00000001_2 (1025_{10}) stored on four consecutive memory words using the (a) little-endian and the (b) big-endian endianness formats.

2.4.3 Arrays on the main memory

In programming languages, an array is a systematic arrangement of similar objects in which each object is identified by an index. A one-dimensional array, a.k.a. a vector, is an array in which the objects, or the array elements, are identified by a one-dimensional index. The following "C" code shows a vector (`V`) that contains four `int` elements and a function that prints the first and the last element of the vector. Notice that the first element is associated with index zero while the last one is associated with index two[12].

```
1 int V[] = {9, 8, 1};
2 void print_V()
3 {
4     printf("First element = %d\n", V[0]);
5     printf("Last  element = %d\n", V[2]);
6 }
```

Vector elements are usually organized in a linear fashion on the memory. Hence, when translating the previous code into machine language, all elements (`int` values) of vector `V` are placed in consecutive memory positions - starting with the first element, *i.e.*, `V[0]`. **The base address of an array is the address of the first memory word that is being used to store the array elements**. Assuming the base address of vector `V` is `000`, then the first element (`V[0]`) is stored starting at memory address `000`. Also, assuming each element requires four memory words[13], the second element

[12]This is a property of the "C" programming language. Other languages, such as Pascal, associate the first element with index one.

[13]The "C" `int` type is used to represent integer numbers and is usually mapped to 32-bit signed numbers.

is stored starting at memory address `004` and the third one starting at memory address `008`. Figure 2.10 illustrates the contents of vector `V` placed on memory starting at address `000`. Notice that, in this example, each element is a 32-bit number that is stored on four consecutive memory words using the little-endian format.

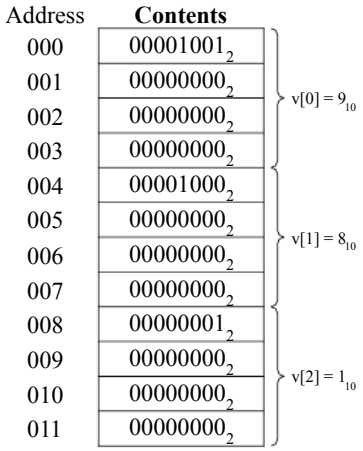


Figure 2.10: Elements of vector `V` stored on memory starting at address `000`.

The previous example showed an array of `int` elements. Nonetheless, in "C", the programmer may also create arrays of other types. For example, one may create an array of `char`, in which each element occupies only one byte, an array of `double`, in which each element occupies 8 bytes, or even an array of a new type defined with the `struct` operator, in which each element may occupy several bytes.

Let:

- V_{addr} be the base address of a vector V;
- $elem_{size}$ be the size of each element of V in bytes;
- $V[i]$ be the i$^{\text{th}}$ element of the vector;

In "C", and several other programming languages, each element $V[i]$ occupies $elem_{size}$ memory words of a byte addressable memory and is placed at the main memory starting at address $\&V[i]$, which is defined by Equation 2.12.

$$\&V[i] = V_{addr} + i \times elemn_{size} \tag{2.12}$$

A multi-dimensional array is an array in which each element is identified by a multi-dimensional index. The following "C" code shows a two-dimensional array (`M`) that contains six `int` elements and a function that prints two elements of the array. In this example, each element is associated with a unique two-dimensional index $[x][y]$ so that $x \in [0\,..\,1]$ and $y \in [0\,..\,2]$.

```
1 int M[][] = { {7, 9, 11},
2               {2, 8, 1} };
3 void print_M()
4 {
5     printf("Element M[0][0] = %d\n", M[0][0]);
6     printf("Element M[1][2] = %d\n", M[1][2]);
7 }
```

Two-dimensional arrays are commonly used to represent matrices. In this context, the first part of the index ($[x]$) is often used to identify the row and the second part ($[y]$) is used to identify the column. Hence, the element located at the first row and last column in previous example is identified by index `M[0][2]`.

The way two-dimensional arrays are organized on memory depends on the programming language. In "C", elements are grouped by row and each row is placed on memory consecutively. Hence, in the previous example, the elements of the first row, *i.e.*, `M[0][0]=7`, `M[0][1]=9`, and `M[0][2]=11`, are placed first on memory. Then, the elements of the second row, *i.e.*, `M[1][0]=2`, `M[1][1]=8`, and `M[1][2]=1`, are placed after the elements of the first row. This way of organizing two-dimensional arrays on memory is known as row-major order.

Let:

- A be a $M \times N$ two-dimensional array in "C", *i.e.*, an array in which the $x \in [0 \,..\, M-1]$ and $y \in [0 \,..\, N-1]$.
- A_{addr} be the base address of array A;
- $elem_{size}$ be the size of each element of A in bytes;
- $A[x][y]$ be the array element associated with index `[x][y]`;

In "C", each element $A[x][y]$ occupies $elem_{size}$ memory words of a byte addressable memory and is placed at the main memory starting at address $\&A[x][y]$, which can be computed using Equation 2.13.

$$\&A[x][y] = A_{addr} + \underbrace{x \times elem_{size} \times N}_{\text{offset 1}} + \underbrace{y \times elem_{size}}_{\text{offset 2}} . \tag{2.13}$$

Notice that Equation 2.13 adds to the base address (A_{addr}) two offsets: offset 1 and offset 2. The first offset is the amount of space in bytes required to store all elements that belong to previous rows, *i.e.*, rows that must be placed before row x. The second offset is the amount of space in bytes required to store all elements that belong to the same row but must be placed before element $A[x][y]$, *i.e.*, the elements that has a column index less than y.

NOTE: The row-major order is used in the following programming languages: "C", "C++", "Objective-C", "PL/I", "Pascal", and other.
Some programming languages, such as "Fortran", "MATLAB", "GNU Octave", "R", "Julia", and other, organize two-dimensional arrays on memory using the column-major order. In this case, elements of a two-dimensional array are organized in memory column by column, *i.e.*, the elements of the first column are placed first then the elements of the second column are placed after the elements of the first one and so on.

2.4.4 Structs on the main memory

In "C", structs are data types defined by the user in which data items of different types may be grouped and combined in a single data type. The following "C" code shows an example in which an `int`, an one-dimensional array with 255 `char` elements and a `short` data item are combined in a struct named `user_id` to form a new data type. Notice that each data item inside the struct is identified by a name, the first one by `id`, the second one by `name`, and the third one by `level`. These items are also known as the "fields" of the struct data type.

```
1 struct user_id {
2         int   id;
3         char  name[256];
4     short level;
5 };
6
7 struct user_id manager;
8
```

```
9  void print_manager()
10 {
11     printf("Manager id    = %d\n", manager.id);
12     printf("Manager name  = %s\n", manager.name);
13     printf("Manager level = %d\n", manager.level);
14 }
```

Assuming each `int`/`char`/`short` item requires four/one/two bytes to be stored on memory, the struct data type defined in the previous example requires 262 bytes to be stored on memory: four bytes to store the `id` field, 256 bytes to store the `name` field and two bytes to store the `level` field.

All fields of a single struct item are stored sequentially on memory in the same order they appear on the declaration. Hence, in the previous example, the field `id` is placed first, then field `name` is placed next and, finally, field `level` is placed last.

The base address of an struct is the address of the first memory word that is being used to store the fields of the struct. Assuming the base address of variable `manager` in previous example is `000`, the field `id` is placed on addresses `000` to `003`, field `name` is placed on addresses `004` to `259`, and field `level` is placed on addresses `260` and `261`. Figure 2.11 illustrates the contents of variable `manager` placed on memory starting at address `000`.

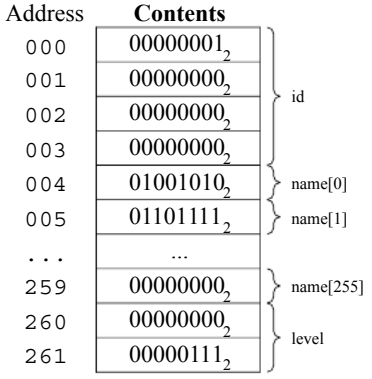


Figure 2.11: Elements of variable `manager` stored on memory starting at address `000`.

2.5 Encoding instructions

Computer instructions are usually encoded as a sequence of bits. The number of bits required to encode each instruction varies accordingly to the computer architecture. For example, at one hand, the RISC-V instruction set architecture defines that all RV32I instructions are encoded using a sequence of 32 bits. On the other hand, instructions that belong to the x86 instruction set architecture family are encoded with a varying number of bits. Figure 2.12 illustrates how two different instructions, that belong to two different instruction set architectures, are encoded.

The 16 bits of the x86 instruction illustrated on Figure 2.12 (a) are organized in four fields: opcode, mod, op1 and op2. The opcode, or operation code, field contains a value that is used by the computer to identify the instruction. In this case, the value 10001001_2 indicates that this is a `mov` instruction. The other fields, mod, op1 and op2, are the instruction parameters. In this case, they specify that the `mov` operation must be performed between registers `%esp` and `%ebp`, which are indicated by values 100_2 and 101_2 on fields op1 and op2.

The 32 bits of the RISC-V instruction illustrated on Figure 2.12 (b) are organized in six fields: funct7, rs2, rs1, funct3, rd, and opcode. The funct7, funct3, and opcode fields contains values that is used by the computer to identify the instruction and, hence, the operation that must be performed. In this case, these values indicate that this is an `add` instruction. The other fields, rs2, rs1, and rd, are the instruction

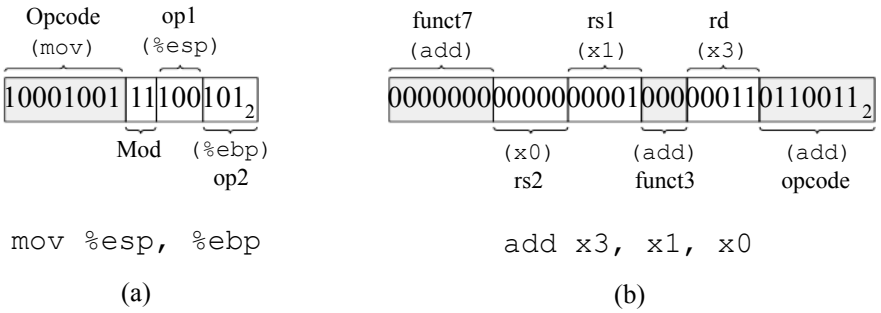


Figure 2.12: Encoding of two different instructions: (a) instruction `mov`, which belongs to the x86 instruction set architecture, and (b) instruction `add`, which belongs to the RISC-V instruction set architecture.

parameters. These parameters specify that the `add` operation must be performed using the values stored in registers `x1` and `x0` and the result stored in register `x3`, which are identified by values 0001_2, 0000_2, and 0011_2 on fields rs1, rs2, and rd, respectively.

Most modern computers store the code, *i.e.*, the program instructions, on the same memory they store the data - the main memory. Also, modern computer instructions are encoded using multiples of 8 bits so that they fit the size of multiple memory words on a byte addressable memory. Figure 2.13 shows an example of how a program written in x86 assembly language is mapped to machine language and stored on a byte addressable memory. Notice that, the first instruction, `push $ebp`, is encoded using one byte while the third one, `imul $113, 12(%ebp), %eax`, is encoded using four bytes. Also, notice that instructions are placed sequentially on memory, in the same order they appear on the original assembly program.

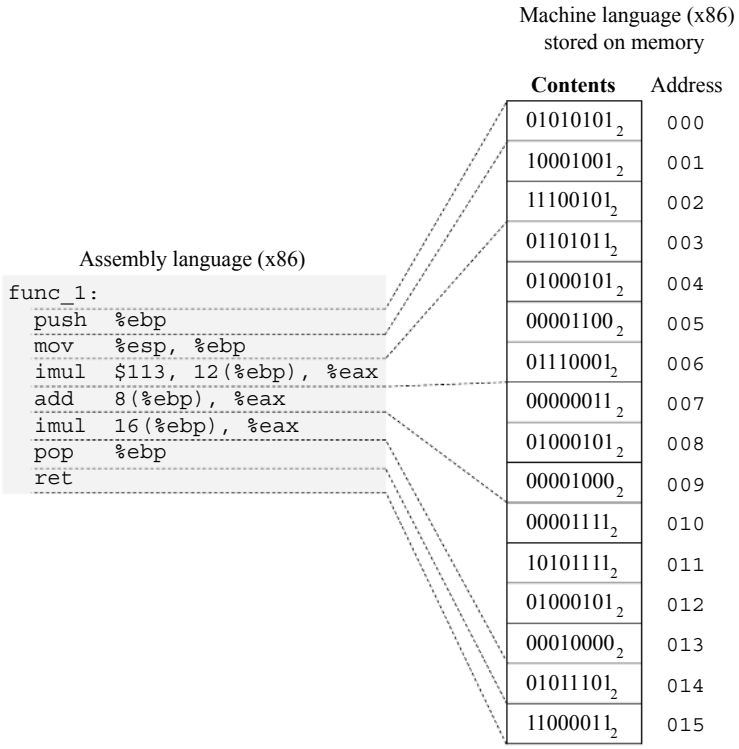


Figure 2.13: Mapping x86 instructions from an assembly language program to memory.

Chapter 3

Assembly, object, and executable files

This chapter presents the main concepts and elements of assembly, object, and executable files.

3.1 Generating native programs

In this section, we discuss how a program written in a high-level language, such as C, is translated into a native program.

As discussed in Chapter 1, **a native program is a program encoded using instructions that can be directly executed by the computer hardware, without help from an emulator or a virtual machine**. These programs are usually automatically translated from programs written in high-level languages, such as C, by tools like compilers, assemblers, and linkers.

A program written in a high-level language, such as C, is encoded as a plain text file. High-level languages are designed to be agnostic of ISA and they are composed of several abstract elements, such as variables, repetition, or loop statements, conditional statements, routines, *etc.*. The following code shows an example of a program written using the C language, which is a high-level language.

```
1 int main()
2 {
3   int r = func (10)
4   return r+1;
5 }
```

A compiler is a tool that translates a program from one language to another. Usually, programming language compilers are employed to translate programs written in high-level languages into lower-level languages. For example, a C compiler is employed to translate a program written in C language into assembly language. The GNU project C and C++ compiler[1], or `gcc`, is a compiler that is capable of translating programs written in C and C++ languages into assembly programs, *i.e.*, a program written in assembly language. The following command line illustrates how the `riscv64-unknown-elf-gcc` tool, a GNU project C and C++ compiler that produces code for RISC-V based computers, can be invoked to produce a RV32I assembly program from a C program. In this example, the C program is stored on the `main.c` file and the RV32I assembly program will be stored on the `main.s` file.

```
$ riscv64-unknown-elf-gcc -mabi=ilp32 -march=rv32i -S main.c -o main.s
```

[1] `https://gcc.gnu.org/`

An assembly program is also a program encoded as a plain text file. The following code shows an example of a program written using the RV32I assembly language. This program has the same semantics as the previous C program.

```
1   .text
2   .align        2
3 main:
4   addi sp,sp,-16
5   li   a0,10
6   sw   ra,12(sp)
7   jal  func
8   lw   ra,12(sp)
9   addi a0,a0,1
10  addi sp,sp,16
11  ret
```

Different from high-level languages, assembly language is very close to the ISA. For example, the previous assembly program contains references to instructions (*e.g.*, `addi`, `li`, ...) and registers (*e.g.*, `sp`, `ra`, `a0`) that belong to the RV32I ISA. Lines 4 to 11 of the previous code contain assembly instructions, which are converted by the assembler into RV32I machine instructions. As a consequence, they are ISA dependent, *i.e.*, an assembly program generated for one ISA is usually not compatible with other ISAs.

Machine language is a low-level language that can be directly processed by a computer's central processing unit (CPU). **An assembler is a tool that translates a program in assembly language into a program in machine language**. For example, it converts assembly instructions (encoded as sequences of ASCII characters) into machine instructions (encoded as sequences of bits accordingly to the ISA). Each assembly language is associated with a given ISA.

The "GNU Assembler"[2] tool, or `as`, is an assembler that is capable of translating programs written in several assembly languages into machine language for their respective ISAs. In this book we will use the `as` tool to translate RV32IM assembly programs to machine language programs. The following command line illustrates how the `riscv64-unknown-elf-as` tool, a version of the GNU Assembler that generates code for RISC-V ISAs, can be invoked to assemble a RV32I assembly program. In this example, the RV32I assembly program is stored on the `main.s` file and the result, a file that contains code in machine language, will be stored on the `main.o` file.

```
$ riscv64-unknown-elf-as -mabi=ilp32 -march=rv32i main.s -o main.o
```

Assemblers usually produce object files that are encoded in binary and contains code in machine language. The object file also contains other information, such as the list of symbols (*e.g.*, global variables and functions) defined in the file. There are several known file formats used to encode object files. The **Executable and Linking Format**, or ELF, is frequently used on Linux-based systems while the **Portable Executable** format, or PE, is used on Windows-based systems. The `riscv64-unknown-elf-as` tool, used in the previous example, produces an ELF-based object file.

Even though the object file produced by the assembler contains code in machine language, it is usually incomplete in the sense that it may still need to be relocated (more on relocation later) or linked with other object files to compose the whole program. For example, the code in an object file may need to be linked with the C library so that the program can invoke the `printf` function. As a consequence, the object file produced by the assembler is not an executable file.

A linker is a tool that "links" together one or more object files and produces an executable file. The executable file is similar to an object file in the

[2] `https://www.gnu.org/software/binutils/`

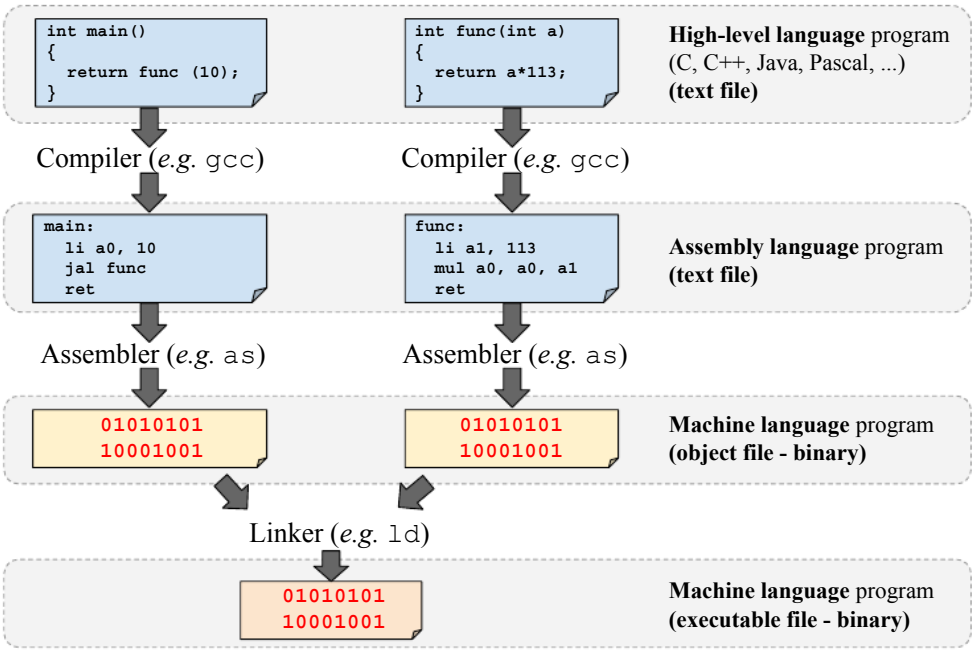


Figure 3.1: Native code generation flow.

sense that it is encoded in binary and contains code in machine language. Nonetheless, it contains all the required elements (*e.g.*, libraries) for execution.

The following command line illustrates how the `riscv64-unknown-elf-ld` tool, a version of the GNU Linker[3] tool that links object files generated for RISC-V ISAs, can be invoked to link two object files together: the `main.o` and `mylib.o` object files. In this example, the linker will produce an executable file named `main.x`.

```
$ riscv64-unknown-elf-ld -m elf32lriscv main.o mylib.o -o main.x
```

Figure 3.1 illustrates the code generation process used to produce a native program executable file from a C program organized in two files. First, the two C program files are translated into assembly programs by the compiler. Then, the assembly programs are assembled by the assembler, which produces object files. Finally, the linker links the object files together producing an executable file.

Assuming the high-level language program files are named `main.c` and `func.c`, the following sequence of commands produce a RV32I executable file named `main.x`.

```
$ riscv64-unknown-elf-gcc -mabi=ilp32 -march=rv32i -S main.c -o main.s
$ riscv64-unknown-elf-as -mabi=ilp32 -march=rv32i main.s -o main.o
$ riscv64-unknown-elf-gcc -mabi=ilp32 -march=rv32i -S func.c -o func.s
$ riscv64-unknown-elf-as -mabi=ilp32 -march=rv32i func.s -o func.o
$ riscv64-unknown-elf-ld -m elf32lriscv main.o func.o -o main.x
```

3.1.1 Inspecting the contents of object and executable files

Object and executable files are encoded as binary files, hence, it is not easy to look at their contents directly. To inspect their contents, developers usually rely on programs that decode and translate their information to a human-readable representation, usually a textual format. There are several tools that can be used to inspect the contents of object and executable files. The `objdump`, `nm`, and `readelf` tools are examples of tools that can be used to inspect the contents of object and executable files on Linux-based systems. The following sections show several examples of how these tools can be used to inspect the contents of object and executable files.

[3] `https://www.gnu.org/software/binutils/`

3.2 Labels, symbols, references, and relocation

3.2.1 Labels and symbols

Labels are "markers" that represent program locations. They are usually defined by a name ended with the suffix ":" and can be inserted into an assembly program to "mark" a program position so that it can be referred to by assembly instructions or other assembly commands, such as directives.

The following code shows an assembly program that contains two labels: `x:`, defined in line 1, and `sum10:`, defined in line 4. The `x:` label identifies a program location that contains a variable, which is allocated and initialized by the directive `.word 10` (line 2). The `sum10:` label identifies the program location that contains the first instruction of the `sum10` routine, in other words, it defines the routine entry point. Also, in this example, the `x:` label is used in instruction `lw` (line 5) to refer to variable `x`.

```
1 x:
2   .word 10
3
4 sum10:
5   lw   a0, x
6   addi a0, a0, 10
7   ret
```

Global variables and program routines are program elements that are stored on the computer main memory. Each variable and each routine occupies a sequence of memory words and are identified by the address of the first memory word they occupy. At one hand, to read the contents of a global variable, or execute a routine, it suffices to have their addresses, *i.e.*, the address of the first memory word they occupy[4] . On the other hand, the addresses assigned to variables and routines are only final on the executable file, after the linker links together the multiple object files into a single file. Hence, assembly programs require a mechanism to refer to global variables and routines. This is accomplished by using labels, as illustrated in the previous example. In this context, before allocating space for each global variable or producing the code for each routine, the programmer (or the compiler) defines a label that will be used to identify the variable or the routine.

Program symbols are "names" that are associated with numerical values and **the "symbol table" is a data structure that maps each program symbol to its value**. Labels are automatically converted into program symbols by the assembler and associated with a numerical value that represents its position in the program, which is a memory address. The assembler adds all symbols to the program's "symbol table", which is also stored on the object file.

We can inspect the contents of the object file by using tools that decode the information on the object file and shows them on a human-readable format, *i.e.*, a textual format. The GNU `nm` tool, for example, can be used to inspect the "symbol table" of an object file. Assuming the previous code was encoded into an object file named `sum10.o`, we can inspect its symbol table by executing the `riscv64-unknown-elf-nm` tool as follows.

```
$ riscv64-unknown-elf-nm  sum10.o
00000004 t .L0
00000004 t sum10
00000000 t x
```

[4]To execute a routine it suffices to set the PC with the address of the first instruction of the routine - this is usually done by executing a "jump" instruction, which sets the PC with a given value.

Notice that, in this case, the symbol table contains three symbols: `.L0`[5], `sum10`, and `x`, which are associated with values `00000004`, `00000004`, and `00000000`, respectively.

The programmer may also explicitly define symbols by using the `.set` directive. The following code shows a fragment of assembly code that employs the `.set` directive to define a symbol named `answer` and assign value 42 to it.

```
1 .set answer, 42
2 get_answer:
3   li a0, answer
4   ret
```

Assuming the previous code is stored on a program file named `get_answer.s`, we can assemble it and inspect the object file symbol table by executing the following commands:

```
$ riscv64-unknown-elf-as -march=rv32im get_answer.s -o get_answer.o
$ riscv64-unknown-elf-nm  get_answer.o
0000002a a answer
00000000 t get_answer
```

Notice that the symbol table contains two symbols: `answer` and `get_answer`. The `answer` symbol is an absolute symbol, *i.e.*, its value is not changed during the linking process – this is indicated by the letter `'a'` on the output. The `get_answer` symbol is a symbol that represents a location on the `.text` section and may have its value (which is an address) changed during the relocation process. The next sections discuss the relocation process and the program sections' concept.

3.2.2 References to labels and relocation

Each reference to a label must be replaced by an address during the assembling and linking processes. For example, in the previous code, the reference to label `x:`, in instruction `lw` (line 5), is replaced by address 0, *i.e.*, the address of the variable represented by label `x`, when assembling the program.

To illustrate this concept, let us consider the following RV32I assembly program, which contains four instructions and two labels. The first label (`trunk42` - line 1) identifies the entry point of a function while the second one (`done` - line 5) identifies a program location that is the target of a branch instruction[6], which is displayed in line 3.

```
1 trunk42:
2   li   t1, 42
3   bge  t1, a0, done
4   mv   a0, t1
5 done:
6   ret
```

When assembling this program, the assembler translates each assembly instruction (*e.g.*, `li`, `bge`, ...) to a machine instruction, *i.e.*, an instruction encoded with 32 bits. As a result, the program occupies a total of 16 memory words, four for each instruction. Also, the assembler maps the first instruction to address `0`, the second one to address `4`, and so on. In this context, the `trunk42` label, which marks the

[5]The `.L0` symbol was automatically introduced by the assembler when translating the `lw a0, x` assembly instruction. This is a special instruction called pseudo-instruction that will be discussed latter on Section 6.4.

[6]A branch instruction is an instruction that change the execution flow under certain conditions - In this example, the `bge t1, a0, done` (branch greater equal) instruction jumps to the position identified by the `done` label if the value in register `t1` is greater or equal to the value in register `a0`.

beginning of the program, is associated with address `0` and the `done` label, which marks the position in which instruction `ret` is located, is associated with address `c`. Since the `bge` instruction has a reference to label `done`, the assembler encodes the address associated with the `done` label (address `c`) in the fields of this instruction.

The GNU `objdump` tool can be used to inspect several parts of the object file. The following example shows how to use the `riscv64-unknown-elf-objdump`[7] tool to decode the data and instructions on the `trunk.o` file so that we can inspect its contents.

```
$ riscv64-unknown-elf-objdump -D trunk.o

trunk.o:     file format elf32-littleriscv

Disassembly of section .text:

00000000 <trunk42>:
   0: 02a00313           li t1,42
   4: 00a35463           bge t1,a0,c <done>
   8: 00030513           mv a0,t1

0000000c <done>:
   c: 00008067           ret

   ...
```

Notice that, for each instruction, it shows its address, its encoding in hexadecimal, and a text that resembles assembly code[8]. The `bge t1, a0, done` instruction, for example, is mapped to address `4` and encoded with the 32-bit value `00a35463`. The `objdump` tool indicates that it refers to the label `done`, which is mapped to address `c` (`bge t1,a0,c <done>`). Also, notice that the labels (and their addresses) are displayed on their respective program position.

In the previous example, the `trunk42` function starts at address `0`, however, when linking this object file (`trunk.o`) with others, the linker may need to move the code (assign new addresses) so that they do not occupy the same addresses. In this process, the addresses associated with labels may change and each reference to a label must also be fixed to reflect the new addresses.

Relocation is the process in which the code and data are assigned new memory addresses. As discussed previously, during the relocation process, the linker needs to adjust the code and data to reflect the new addresses. More specifically, the addresses associated with labels on the symbol table and the references to labels must be adjusted. **The relocation table is a data structure that contains information that describes how the program instructions and data need to be modified to reflect the addresses reassignment**. Each object file contains a relocation table and the linker uses their information to adjust the code when performing the relocation process.

The following example shows how to use the `riscv64-unknown-elf-objdump` tool to inspect the contents of the relocation table on the `trunk.o` file. Notice that, in this case, the object file contains one relocation record, which indicates that the instruction on address `4`, a RISC-V branch instruction, contains a reference to label `done`. The linker uses this information to adjust the branch instruction when the `done` label is assigned a new address.

```
$ riscv64-unknown-elf-objdump -r trunk.o

trunk.o:     file format elf32-littleriscv
```

[7] The `-D` option instructs the `riscv64-unknown-elf-objdump` tool to disassemble the contents of the object file.

[8] Even though the syntax is similar, the code showed by the `objdump` tool is not assembly code.

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000004 R_RISCV_BRANCH    done
```

The following example shows what happens to the previous program once it is linked. First, we produce the `trunk.x` file by invoking the linker. Then, we inspect the contents of the `trunk.x` file by using the `riscv64-unknown-elf-objdump` tool.

```
$ riscv64-unknown-elf-ld -m elf32lriscv trunk.o -o trunk.x
$ riscv64-unknown-elf-objdump -D trunk.x

trunk.x:     file format elf32-littleriscv

Disassembly of section .text:

00010054 <trunk42>:
   10054: 02a00313           li t1,42
   10058: 00a35463           bge t1,a0,10060 <done>
   1005c: 00030513           mv a0,t1

00010060 <done>:
   10060: 00008067           ret
   ...
```

Notice that the code on the `trunk.x` program was relocated, *i.e.*, assigned new addresses. In this example, the code of the `trunk42` routine starts at address `10054` and the `bge` instruction jumps to address `10060` in case the value in register `t1` is greater or equal to the value in register `a0`.

3.2.3 Undefined references

As discussed in previous sections, assembly code relies on labels to refer to program locations. In some cases, an assembly code refers to a label that is not defined in the same file. This is common when invoking a routine that is implemented on another file or when accessing a global variable that is declared on another file. The following example shows an assembly code that refers to a label that is not defined on the same file, called `main.s`. The `exit` label is used on instruction `jal` (line 4), however, it is not defined in this file[9].

```
1 # Contents of the main.s file
2 start:
3   li  a0, 10
4   li  a1, 20
5   jal exit
```

The assembler assembles this program and register the `exit` label on the symbol table as an undefined symbol. The `riscv64-unknown-elf-nm` tool identifies the undefined symbols by placing the 'U' character before the symbol name. Assuming the previous code was assembled into the `main.o` object file, we can inspect the contents of its symbol table as follows:

```
$ riscv64-unknown-elf-nm main.o
00000000 t start
         U exit
```

The assembler also register the reference to this symbol on the relocation table. The `riscv64-unknown-elf-objdump` tool shows that the `main.o` object file includes a relocation record for the reference to the `exit` label on the `jal` instruction.

[9]The `jal` instruction is used to invoke routines. In this case, it is invoking the `exit` routine. This instruction will be further discussed in Section 6.7.3.

```
$ riscv64-unknown-elf-objdump -r main.o

main.o:     file format elf32-littleriscv

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000008 R_RISCV_JAL       exit
```

When linking the object files, the linker must resolve the undefined symbols, *i.e.*, it must find the symbol definition and adjust the symbol table and the code with the symbol value. In the previous example, the linker will look for the `exit` symbol so that it can adjust the `jal` instruction to refer to the correct address. In case it cannot find the definition of the symbol, it stops the linking process and emits an error message. The following example illustrates this situation. In this case, we are trying to link the `main.o` file without providing another object file that contains a definition of the `exit` label. Notice that the linker emits the error message `undefined reference to ‘exit’`.

```
$ riscv64-unknown-elf-ld -m elf32lriscv main.o -o main.x
riscv64-unknown-elf-ld: warning: cannot find entry symbol start; ...
riscv64-unknown-elf-ld: main.o: in function ‘start’:
(.text+0x8): undefined reference to ‘exit’
```

3.2.4 Global vs local symbols

Symbols are classified as local or global symbols. Local symbols are only visible on the same file, *i.e.*, the linker does not use them to resolve undefined references on other files. Global symbols, on the other hand, are used by the linker to resolve undefined reference on other files.

By default, the assembler registers labels as local symbols. The `.globl` directive is an assembly directive that instructs the assembler to register a label as a global symbol. The following example shows an assembly program in which the `.globl` directive is used to instruct the assembler to register the `exit` label as a global symbol on the symbol table.

```
1 # Contents of the exit.s file
2 .globl exit
3 exit:
4   li a0, 0
5   li a7, 93
6   ecall
```

Assuming the code that invokes the `exit` function is located on the `main.s` file and the `exit` function is located on the `exit.s` file, the following sequence of commands shows how to assemble both files and link them together.

```
$ riscv64-unknown-elf-as -march=rv32im main.s -o main.o
$ riscv64-unknown-elf-as -march=rv32im exit.s -o exit.o
$ riscv64-unknown-elf-ld -m elf32lriscv main.o exit.o -o main.x
riscv64-unknown-elf-ld: warning: cannot find entry symbol start; ...
```

Notice that the linker is not producing the `undefined reference to ‘exit’` anymore. It is worth noting that, in case the `exit` label is not registered as a global label, the linker will not use it to resolve the undefined symbol and the linking process will fail.

3.2.5 The program entry point

Every program has an entry point, *i.e.*, the point from which the CPU must start executing the program. The entry point is defined by an address, which is the address of the first instruction that must be executed.

The executable file has a header that contains several information about the program and one of the header fields store the entry point address. In this context, once the operating system loads the program into the main memory, it sets the PC with the entry point address so the program starts executing.

The linker is responsible for setting the entry point field on the executable file. To do so, it looks for a symbol named `start`. If it finds it, it sets the entry point field with the `start` symbol value. Otherwise, it sets the entry point to a default value, usually the address of the first instruction of the program.

To select the program entry point, the programmer (or the compiler) may define the label `start` right before the first instruction that must be executed. In the previous example we intentionally used the `start` label on the `main.s` assembly program to mark the program entry point. Nonetheless, as indicated by the warning message (`warning: cannot find entry symbol start; defaulting to ...010054`), the linker was not able to find the entry symbol. It happened because the `start` label must be registered as a global symbol for the linker to recognize it as the entry point. The following code shows how the `main.s` file can be adjusted to register the `start` label as a global symbol.

```
1 # Contents of the main.s file
2 .globl start
3 start:
4   li  a0, 10
5   li  a1, 20
6   jal exit
```

Once the `start` label is registered as a global symbol, the linker uses its address to set the entry point information. The following sequence of commands assemble the `main.s` and the `exit.s` assembly programs and link them together into the `main.x` executable file. In this case, there were no error nor warning messages because we used the `.globl` directive to set both the `exit` and the `start` labels as global symbols, allowing the linker to resolve the exit reference and set the program entry point.

```
$ riscv64-unknown-elf-as -march=rv32im main.s -o main.o
$ riscv64-unknown-elf-as -march=rv32im exit.s -o exit.o
$ riscv64-unknown-elf-ld -m elf32lriscv exit.o main.o -o main.x
```

Notice that when invoking the linker, in this case, we passed the `exit.o` object file before the `main.o` file. Because of this, the linker places the contents of the `exit.o` file before the contents of the `main.o` file on the `main.x` file. This can be observed by listing the contents of the `main.x` file with the `riscv64-unknown-elf-objdump` tool, as follows:

```
$ riscv64-unknown-elf-objdump -D main.x

main.x:     file format elf32-littleriscv

Disassembly of section .text:

00010054 <exit>:
   10054: 00000513           li a0,0
   10058: 05d00893           li a7,93
   1005c: 00000073           ecall

00010060 <start>:
```

```
10060: 00a00513          li a0,10
10064: 01400593          li a1,20
10068: fedff0ef          jal ra,10054 <exit>
...
```

Even though the linker placed the `exit` function first, the code associated with the `start` label will be executed first because the entry point field contains the address associated with the `start` label.

The GNU `readelf` tool can be used to display information about ELF files. The following command shows how the `riscv64-unknown-elf-readelf` tool can be used to inspect the header of the `main.x` executable file. Notice that the entry point address field was set to `0x10060`, *i.e.*, the address of the `start` label.

```
$ riscv64-unknown-elf-readelf -h main.x
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  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:               0x10060
  Start of program headers:          52 (bytes into file)
  Start of section headers:          476 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         1
  Size of section headers:           40 (bytes)
  Number of section headers:         6
  Section header string table index: 5
```

3.3 Program sections

Executable and object files, and assembly programs are usually organized in sections. A section may contain data or instructions, and the contents of each section are mapped to a set of consecutive main memory addresses. The following sections are often present on executable files generated for Linux-based systems:

- `.text`: a section dedicated to store the program instructions;
- `.data`: a section dedicated to store initialized global variables, *i.e.*, the variables that need their value to be initialized before the program starts executing;
- `.bss`: a section dedicated to store uninitialized global variables;
- `.rodata`: a section dedicated to store constants, *i.e.*, values that are read by the program but not modified during execution.

When linking multiple object files, the linker groups information from sections with the same name and places them together into a single section on the executable file. For example, when linking multiple object files, the contents of the `.text` sections from all object files are grouped together and placed sequentially on the executable file on a single section that is also called `.text`. Figure 3.2 shows the layout of an RV32I executable file that was generated by the `riscv64-unknown-elf-ld` tool, and is encoded using the Executable and Linking Format. This file contains three sections: the `.data`, the `.rodata`, and the `.text` sections. The contents of section `.text` are

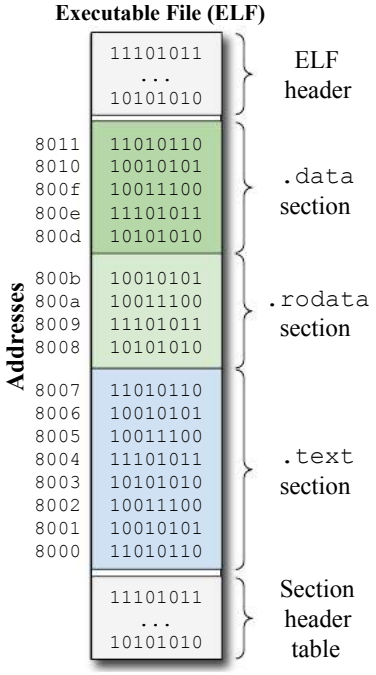


Figure 3.2: Layout of an executable file encoded using the ELF.

mapped to addresses `8000` to `8007`, while the contents of section `.data` are mapped to addresses `800d` to `8011`.

By default, the GNU assembler tool adds all the information to the `.text` section. To instruct the assembler to add the assembled information into other sections, the programmer (or the compiler) may use the `.section secname` directive. This directive instructs the assembler to place the following assembled information into the section named `secname`. The following example illustrates how the `.section` directive can be used to add instructions to the `.text` section and variables to the `.data` section.

```
1  .section .data
2  x: .word 10
3  .section .text
4  update_x:
5    la t1, x
6    sw a0, (t1)
7    ret
8  .section .data
9  y: .word 12
10 .section .text
11 update_y:
12   la t1, y
13   sw a0, (t1)
14   ret
```

The `.section .data` directive in the first line of the previous example is instructing the assembler to add information to the `.data` section from this point on. The second line contains a label (`x:`) and a `.word` directive, which are used together to declare and initialize a global variable named `x`. The `.section .text` directive in the third line instructs the assembler to add the following information to the `.text` section. As a consequence, the `update_x` label (line 4) refers to a position on the `.text` section and the next three instructions (lines 5-7) are added at the `.text` section. The `.section .data` directive in line eight instructs the assembler to add the following

information to the `.data` section. Hence, the `y` variable, created by combining the `y:` label with the `.word` directive, is added to the `.data` section, right after the `x` variable. After this, `.section .text` directive in line 10 instructs the assembler to add the next information to the `.text` section. Finally, the `update_y` label (line 11) refers to a position in the `.text` section and the remaining instructions (lines 12-14) are added to the `.text` section.

Assuming the previous code is stored on a file named `prog.s`, we may assemble the program and inspect the contents of the object file using the following commands[10]:

```
$ riscv64-unknown-elf-as -march=rv32im prog.s -o prog.o
$ riscv64-unknown-elf-objdump -D prog.o

prog.o:     file format elf32-littleriscv

Disassembly of section .text:

00000000 <update_x>:
   0: 00000317           auipc t1,0x0
   4: 00030313           mv t1,t1
   8: 00a32023           sw a0,0(t1) # 0 <update_x>
   c: 00008067           ret

00000010 <update_y>:
  10: 00000317           auipc t1,0x0
  14: 00030313           mv t1,t1
  18: 00a32023           sw a0,0(t1) # 10 <update_y>
  1c: 00008067           ret

Disassembly of section .data:

00000000 <x>:
   0: 000a                 c.slli zero,0x2
...

00000004 <y>:
   4: 000c                 0xc
...
```

Notice that the program routines, represented by the `update_x` and `update_y` labels, and the program instructions are all located on the `.text` section while the global variables, represented by the `x` and `y` labels, and the 32-bit values `000a` and `000c`, are located on the `.data` section. Also, notice that the elements of each section are assigned addresses starting at zero. However, since instructions and data are stored on the same memory, the main memory, we may not load the variables and the instruction into the same memory addresses. The linker prevents this problem by relocating the instructions and data so they are assigned non-conflicting addresses. The following commands show how to invoke the linker to produce an executable file named `prog.x` and how to invoke the `objdump` tool to inspect its contents:

```
$ riscv64-unknown-elf-ld -m elf32lriscv prog.o -o prog.x
$ riscv64-unknown-elf-objdump -D prog.x

prog.x:     file format elf32-littleriscv

Disassembly of section .text:
```

[10] In this example, the `la` assembly instruction is a pseudo-instruction that is converted by the assembler into two machine instructions: `auipc` and `mv`. These two instructions will be later updated by the linker so that they load the address of the respective label into the target register, *i.e.*, register `t1`.

```
00010074 <update_x>:
   10074: 00001317          auipc t1,0x1
   10078: 01c30313          addi t1,t1,28 # 11090 <__DATA_BEGIN__>
   1007c: 00a32023          sw a0,0(t1)
   10080: 00008067          ret

00010084 <update_y>:
   10084: 80418313          addi t1,gp,-2044 # 11094 <y>
   10088: 00a32023          sw a0,0(t1)
   1008c: 00008067          ret

Disassembly of section .data:

00011090 <__DATA_BEGIN__>:
   11090: 000a                c.slli zero,0x2
...

00011094 <y>:
   11094: 000c                0xc
...
```

Notice that the linker assigned addresses `10074` to `1008f` to the contents of the `.text` section and addresses `10090` to `10097` to the contents of the `.data` section.

> **NOTE:** Some operating systems configure the hardware to prevent instructions from writing to memory addresses assigned to the `.text` and `.rodata` sections. Hence, variables should not be placed on these sections. Also, some operating systems configure the hardware to prevent the CPU from executing instructions from memory addresses that are not assigned to the `.text` section. Therefore, it is important to keep the program instructions on the `.text` section.

3.4 Executable vs object files

The Executable and Linking Format, or ELF, is used by several Linux-based operating systems to encode both object and executable files. Even though object and executable files may contain machine code and both can be encoded using the ELF, they differ in the following aspects:

- Addresses on object files are not final and elements from different sections may be assigned the same addresses, as discussed in Section 3.3. As a consequence, the elements of different sections may not reside in the main memory at the same time;
- Object files usually contain several references to undefined symbols, which are expected to be resolved by the linker;
- Object files contain a relocation table so that instructions and data on object files can be relocated on linking. Addresses on executable files are usually final;
- Object files do not have an entry point;

Chapter 4

Assembly language

Assembly programs are encoded as plain text files and contain four main elements:

- **Comments**: comments are textual notes that are often used to document information on the code, however, they have no effect on the code generation and the assembler discards them;
- **Labels**: as discussed in Section 3.2.1, labels are "markers" that represent program locations. They are usually defined by a name ended with the suffix ":" and can be inserted into an assembly program to "mark" a program position so that it can be referred to by assembly instructions or other assembly commands, such as assembly directives;
- **Assembly instructions**: Assembly instructions are instructions that are converted by the assembler into machine instructions. They are usually encoded as a string that contains a mnemonic and a sequence of parameters, known as operands. For example, the "`addi a0, a1, 1`" string contains the `addi` mnemonic and three operands: `a0`, `a1`, and `1`;
- **Assembly directives**: Assembly directives are commands used to coordinate the assembling process. They are interpreted by the assembler. For example, the `.word 10` directive instructs the assembler to assemble a 32-bit value (10) into the program. Assembly directives are usually encoded as strings that contains the directive name, which have a dot ('.') prefix, and its arguments.

As discussed before, comments have no effect on the assembling process and are discarded by the assembler. This is usually performed by a preprocessor, which removes all comments and extra white spaces. Once comments and extra white spaces are discarded, the assembly program contains only three kinds of elements: labels, assembly instructions and assembly directives. Assuming `<label>`, `<instruction>`, and the `<directive>` represent valid labels, assembly instructions, and assembly directives, respectively, the following regular expression can be used to summarize the syntax of the assembly language once its comments and extra white spaces are removed.

```
PROGRAM -> LINES
LINES   -> LINE ['\n' LINES]
LINE    -> [<label>] [<instruction>] |
           [<label>] [<directive>]
```

The first two rules of the previous regular expression indicate that an assembly program is composed by one or more lines, which are delimited by the end of line character, *i.e.*, '`\n`'. The last rule implies that:

- a line may be empty. Notice that the `<label>`, the `<instruction>`, and the `<directive>` elements are optional[1];

[1] Elements expressed between brackets on a regular expression are optional.

- a line may contain a single label;
- a line may contain a label followed by an assembly instruction;
- a line may contain a single assembly instruction;
- a line may contain a label followed by an assembly directive;
- a line may contain a single assembly directive;

The following RV32I assembly code contains examples of valid assembly lines:

```
  x:

sum:  addi a0, a1, 1
      ret
.section  .data
y:    .word 10
```

The following RV32I assembly code contains examples of invalid assembly lines. The first line contains two labels and the second one contains an instruction followed by a label (notice that the label has to precede the instruction when both are located in the same line). The third line contains two instructions and the fourth one contains two assembly directives, however, only one instruction or one directive is allowed per line. The fifth line contains an assembly directive followed by a label, however, the label has to precede the directive when both are located in the same line. The sixth line contains an instruction and an assembly directive on the same line while the seventh line contains an invalid directive.

```
  x: z:
addi a0, a1, 1 sum:
  li a0, 2    li a1, 1
.word 10 .word 20
    .word 10  y:
addi a0, a1, 1  .word 12
.sdfoiywer 1
```

The following RV32I assembly code is also invalid because all elements of a single instruction, *i.e.*, its mnemonic and operands, must be expressed in the same line. This is also a requirement for assembly directives.

```
addi
a0, a1, 1
```

The GNU assembler tool is actually a family of assemblers. It includes assemblers for several ISAs, including the RV32I ISA. Even though the assembly language processed by each GNU assembler is different (usually due to differences on the assembly instructions, which are designed to be similar to the machine instructions on the ISA), most of them share the same syntax for comments and labels, and the same syntax for most of the assembly directives. In this sense, once you learn one assembly language, it should be easy to learn new assembly languages.

In this book we will focus on the RV32I assembly language interpreted by the GNU assembler tool. The following sections discuss the syntax of comments, labels, assembly instructions, and assembly directives on RV32I assembly programs for the GNU assembler tool.

4.1 Comments

RV32I assembly programs may contain line or multi-line comments. On GNU assemblers, line comments are delimited by a line comment character, which is target specific, *i.e.*, it depends on the target ISA. The RV32I GNU assembler uses the `#` character as the line comment character. All characters located between the first occurrence of the line comment character (*e.g.*, `#`) on the line and the end of the same line are considered part of the comment. The following assembly code shows examples of line comments.

```
x: .word 10     # This is a comment
foo:            # My special function
 addi a0, a1, 1 # Adds one to a1 and store on a0 #
# This is # another # comment ## #
```

As comments are discarded by the assembler preprocessor, the previous code is semantically equivalent to the following code.

```
x: .word 10
foo:
 addi a0, a1, 1
```

A multi-line comment is a comment that spans multiple lines. On GNU assemblers, they are delimited by the pair `/*` and `*/`.

```
sum1:
/* This
 is
    a
      multi-line
        comment.
*/
  addi a0, a1, 1
  ret
```

The previous code is semantically equivalent to the following code:

```
sum1: addi a0, a1, 1
      ret
```

4.2 Assembly instructions

Assembly instructions are instructions that are converted by the assembler into machine instructions. They are usually encoded as a string that contains a mnemonic and a sequence of parameters, known as operands. For example, the assembly instruction "`add x10, x11, x12`", which is encoded as plain text using 17 bytes (one for each character), is converted by the assembler into its corresponding machine instruction, which is encoded in four bytes as "`0x00c58533`".

A **pseudo-instruction** is an assembly instruction that does not have a corresponding machine instruction on the ISA, but can be translated automatically by the assembler into one or more alternative machine instructions to achieve the same effect. As an example, the no operation instruction, or "`nop`", is a RV32I pseudo-instruction that is converted by the assembler into the "`addi x0, x0, 0`" instruction[2]. Another

[2]This instruction adds zero to zero and discards the results by storing it on register x0, which is hard-wired to zero.

example is the "`mv`" instruction, which copies the contents of one register into another. In this case, the pseudo-instruction "`mv a5, a7`", which copies the contents of `a7` into `a5`, is converted into the instruction "`addi a5, a7, 0`", which adds zero to the value in `a7` and stores the result on register `a5`.

Appendix A presents a list with most of the RV32I assembly instructions, and the chapters in Part II discuss how these instructions can be used to implement program structures, including conditional sentences, loops, and routines.

The operands of assembly instructions may contain:

- A register name: a register name identifies one of the ISA registers. RV32I ISA registers are numbered from 0 to 31 and are named `x0`, `x1`, ..., `x31`. RV32I registers may also be identified by their aliases, for example, `a0`, `t1`, `ra`, *etc.*. Appendix A presents a list of RV32I registers and their respective aliases.
- An immediate value: an immediate value is a constant that is directly encoded into the machine instruction as a sequence of bits.
- A symbol name: symbol names identify symbols on the symbol table and are replaced by their respective values during the assembling and linking processes. They may identify, for example, symbols that were explicitly defined by the user or symbols created automatically by the assembler, such as the symbols created for labels. Their value are also encoded into the machine instruction as a sequence of bits.

4.3 Immediate values

Immediate values are represented on assembly language by a sequence of alphanumeric characters. Sequences started with the "`0x`" and the "`0b`" prefixes are interpreted as hexadecimal and binary numbers, respectively. Octal numbers are represented by a sequence of numeric digits starting with digit "`0`". Sequences of numeric digits starting with digits "`1`" to "`9`" are interpreted as decimal numbers.

Alphanumeric characters represented between single quotation marks are converted to numeric values using the ASCII table. For example, the `'a'` operand is converted into value ninety seven. The following code shows examples of instructions that use immediate values as operands:

```
1 li a0, 10     # loads value ten into register a0
2 li a1, 0xa    # loads value ten into register a1
3 li a2, 0b1010 # loads value ten into register a2
4 li a3, 012    # loads value ten into register a3
5 li a4, '0'    # loads value forty eight into register a4
6 li a5, 'a'    # loads value ninety seven into register a5
```

To denote a negative integer, it suffices to the '`-`' prefix. For example:

```
1 li a0, -12     # loads value minus twelve into register a0
2 li a1, -0xc    # loads value minus twelve into register a1
3 li a2, -0b1100 # loads value minus twelve into register a2
4 li a3, -014    # loads value minus twelve into register a3
5 li a4, -'0'    # loads value minus forty eight into register a4
6 li a5, -'a'    # loads value minus ninety seven into register a5
```

4.4 Symbol names

Program symbols are "names" that are associated with numerical values and the "symbol table" is a data structure that maps each program symbol to its value. Labels

are automatically converted into symbols by the assembler. Also, the programmer, or the compiler, may explicitly create symbols by using the `.set` directive.

Symbol names are defined by a sequence of alphanumeric characters and the underscore character (_). However, the first character may not be a numeric character. The following names are examples of valid symbol names: `x`, `var1`, `z12345`, `_x`, `_`, `_1`, `_123`, and `_a12b`.

The following names are examples of invalid symbol names: `1`, `1var`, `z@12345`, `x-y`, `-var`, and `a+b`.

The following code shows examples of instructions that use symbol names as operands (lines 4 and 5). The `.set` directive (line 1) creates the `max_temp` symbol and associates value 100 with it. The load immediate instruction (line 4), loads the value of symbol `max_temp` into register `t1`. The branch less equal instruction (`ble`) jumps to the code position represented by symbol `temp_ok` (which is defined automatically by label `temp_ok:`) if the value in register `a0` is less or equal to the value in register `t1`.

```
1 .set max_temp, 100       # Set the max_temp limit
2
3 check_temp:              # check_temp routine
4   li   t1, max_temp      # Loads the max_temp limit into t1
5   ble  a0, t1, temp_ok   # If a0 <= max_temp, then ok
6   jal  alarm             # Else, invokes the alarm routine
7   temp_ok:
8   ret                    # Returns from the routine
```

4.5 Labels

As discussed in Section 3.2.1, labels are "markers" that represent program locations. They can be referred to by instructions and assembly directives and are translated to addresses during the assembling and linking processes.

GNU assemblers usually accept two kinds of labels: symbolic and numeric labels. Symbolic labels are stored as symbols in the symbol table and are often used to identify global variables and routines. They are defined by an identifier followed by a colon (`:`). The identifier follows the same syntax of symbol names, as defined in the previous section. The following code contains two symbolic labels: `age:` and `get_age:`.

```
1 age: .word 42
2
3 get_age:
4   la t1, age
5   lw a0, (t1)
6   ret
```

Numeric labels are defined by a single decimal digit followed by a colon (`:`). They are used for local reference and are not included in the symbol table of executable files. Also, they can be redefined repeatedly in the same assembly program.

References to numeric labels contain a suffix that indicates whether the reference is to a numeric label positioned before ('`b`' suffix) or after ('`f`' suffix) the reference. The following code contains examples of numeric labels and references to them. This code has one symbolic label (`pow`) and two numeric labels (both named `1:`). The first numeric label, located at line 7, marks the beginning of a sequence of instructions that belongs to a loop. The jump instruction located at line 11 jumps back to this label – notice the reference `1b`, which refers to the numeric label '`1`' positioned before the reference. The second numeric label, located at line 12, marks the location of the instruction that is positioned after the loop, *i.e.*, the instruction that must be

executed when the execution flow leaves the loop. The instruction at line 8 jumps to this numeric label when the value of register `a1` is equal to zero – notice the reference `1f`, which refers to the numeric label '`1`' positioned after the reference.

```
1  # Pow function -- computes a^b
2  # Inputs: a0=a, a1=b
3  # Output: a0=a^b
4  pow:
5    mv   a2, a0     # Saves a0 in a2
6    li   a0, 1      # Sets a0 to 1
7  1:
8    beqz a1, 1f     # If a1 = 0 then done
9    mul  a0, a0, a2 # Else, multiply
10   addi a1, a1, -1 # Decrements the counter
11   j    1b         # Repeat
12 1:
13   ret
```

4.6 The location counter and the assembling process

The location counter is an internal assembler counter that keeps track of addresses when a program is being assembled. More specifically, it keeps the address of the next available memory position. Each section has its own location counter, and the active location counter is the location counter of the active section.

To discuss how the location counter is used and updated throughout the assembling process, we will assemble the following assembly program step by step:

```
1  sum42:
2    addi a0, a0, 42
3    ret
```

Upon start, the GNU assembler clears the contents of the sections and the symbol table, initializes all location counters with zero, and selects the `.text` section as the active section. Figure 4.1 illustrates the status of the internal assembler structures upon start.

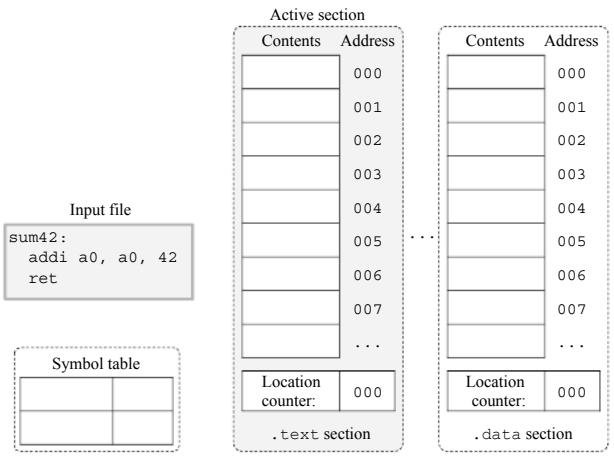


Figure 4.1: Assembler internal structures upon start. The sections' contents and the symbol table are cleared, the location counters are initialized with zero, and the `.text` section is set as the active section.

Once the internal structures are initialized, the assembler reads the input assembly file sequentially, line by line, processing the labels, assembly instructions, and assembly directives one by one, in the order they appear.

The first element in our assembly program is a label named "`sum42:`". When processing a label, the assembler registers it as a symbol at the symbol table and associates it with an address that represents the current program location. The current location is indicated by the active location counter. Figure 4.2 illustrates how the symbol table is updated when the "`sum42:`" label is processed by the assembler. Notice that the assembler register the name `sum42` into the symbol table (1) and associates it with address zero (2), *i.e.*, the address of the active location counter.

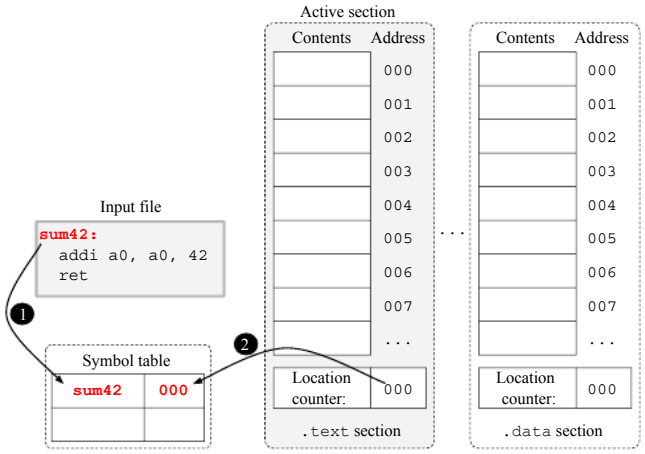


Figure 4.2: Example of a label being processed by the assembler.

The next element in our assembly program is the `addi a0, a0, 42` assembly instruction. In this case, the assembler (1) translates it into a machine instruction, (2) adds it to the active section in the address indicated by the active location counter, and (3) updates the active location counter so it points to the next available address. In this case, the active location counter is incremented by four units because the RV32I instruction that was added to the active section requires four memory words. Figure 4.3 illustrates this process.

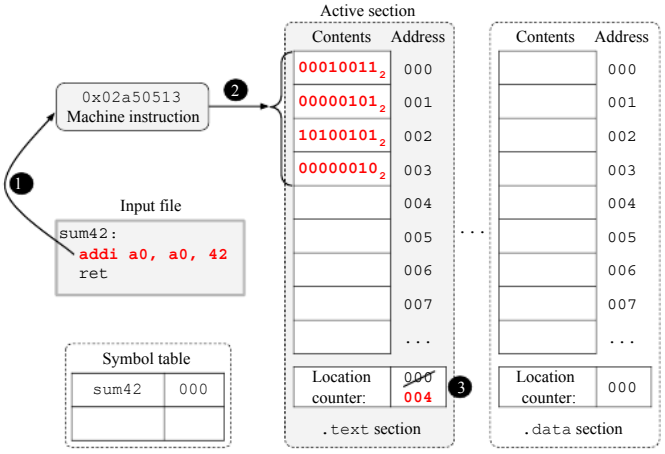


Figure 4.3: Example of an assembly instruction being processed by the assembler.

The last element in our assembly program is also an assembly instruction. Again, the assembler (1) translates it into a machine instruction, (2) adds it to the active section in the address indicated by the active location counter, and (3) updates the active location counter so it points to the next available address. This process is illustrated in Figure 4.4.

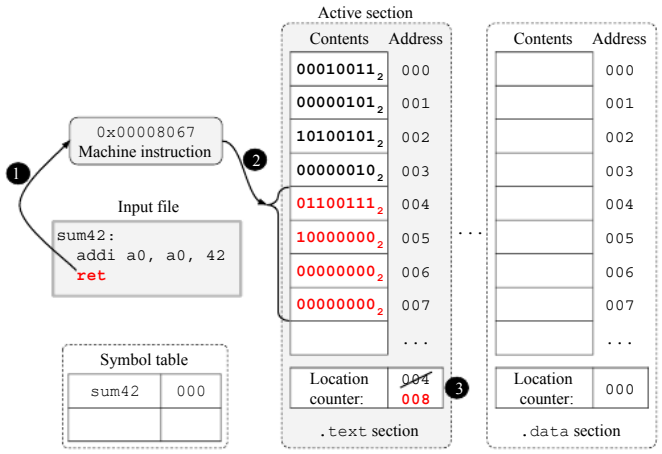


Figure 4.4: Processing the last assembly instruction.

Once all elements from the input file are processed, the assembler stores the section contents, the symbol table, and other relevant information (such as the relocation records), on the object file.

4.7 Assembly directives

Assembly directives are commands used to control the assembler. For example, the `.section .data` directive instructs the assembler to turn the `.data` section into the active section, and the `.word 10` directive instructs the assembler to assemble a 32-bit value (10) and add it to the active section.

Assembly directives are usually encoded as a string that contains the directive name and its arguments. On GNU assemblers, directive names contain a dot ('.') prefix. The next sections discuss some of the most common directives used to implement assembly programs.

4.7.1 Adding values to the program

Table 4.1 contains a list of assembly directives that can be used to add values to a program.

Directive	Arguments	Description
`.byte`	expression [, expression]*	Emit one or more 8-bit comma separated words
`.half`	expression [, expression]*	Emit one or more 16-bit comma separated words
`.word`	expression [, expression]*	Emit one or more 32-bit comma separated words
`.dword`	expression [, expression]*	Emit one or more 64-bit comma separated words
`.string`	string	Emit NULL terminated string
`.asciz`	string	Emit NULL terminated string (alias for .string)
`.ascii`	string	Emit string without NULL character

Table 4.1: List of assembly directives that can be used to add values to a program.

All directives on Table 4.1 add values to the active section. The `.byte`, `.half`, `.word`, and `.dword` directives add one or more values to the active section. Their arguments may be expressed as immediate values, as discussed in Section 4.3, symbols, which are replaced by their value during the assembling and linking processes, or by

arithmetic expressions that combine both. The following code shows examples of valid arguments for these directives. The `.byte` in the first line adds four 8-bit values to the active section (10, 12, 97, and 10). The `.word` directive in the second line adds a 32-bit value associated with symbol `x` to the active section. Notice that value associated with symbol `x` is the address assigned to label `x:`. The `.word` directive in the third line also adds a 32-bit value to the active section, however, in this case, the value is computed by adding four to the value associated with symbol `y`, which is the address assigned to label `y:`.

```
1 x: .byte 10, 12, 'A', 5+5
2 y: .word x
3 z: .word y+4
4 i: .word 0
5 j: .word 1
```

The `.string`, `.asciz`, and `.ascii` directives add strings to the active section. The string is encoded as a sequence of bytes as discussed on Section 2.3. The `.string` and `.asciz` directives also adds, after the string, an extra byte with value zero. They are useful to add NULL-terminated strings to the program[3].

To illustrate the use of the previous directives, let us assemble the following program, which adds values to the `.data` section:

```
1 .section .data
2 msg: .ascii "hello"
3 x:   .word 10
```

As discussed in Section 4.6, the GNU assembler first clears the contents of the sections and the symbol table, initializes all location counters with zero, and selects the `.text` section as the active section. Then, it starts processing the input file. The first assembly element in the input file is the `.section .data` directive, which instructs the assembler to make the `.data` the active section. Figure 4.5 illustrates this process.

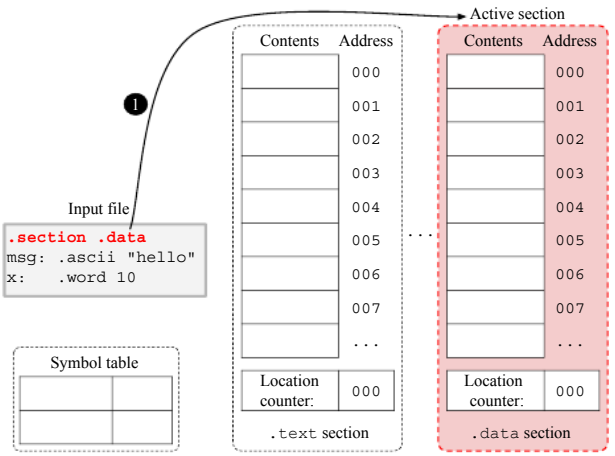


Figure 4.5: Processing the "`.section .data`" directive.

The next element in our assembly program is a label named "`msg:`". In this case, the assembler (1) registers the symbol named `msg` at the symbol table and (2) associates it with an address that represents the current program location, which is indicated by the active location counter, *i.e.*, the location counter of the `.data` section. Figure 4.6 illustrates this process.

[3]Strings declared in C programs are NULL-terminated strings.

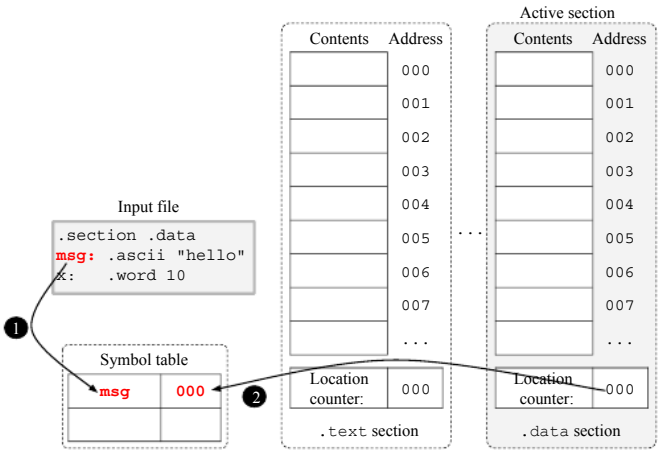


Figure 4.6: Processing the "`msg:`" label.

The next element in our assembly program is the `.ascii "hello"` directive, which instructs the assembler to add a string to the active section. Assuming our input file is encoded using the ASCII standard, the assembler (1) encodes the string as a sequence of bytes based on the ASCII standard, (2) add these bytes to the next available addresses on the active section, and (3) updates the location counter. Figure 4.7 illustrates this process.

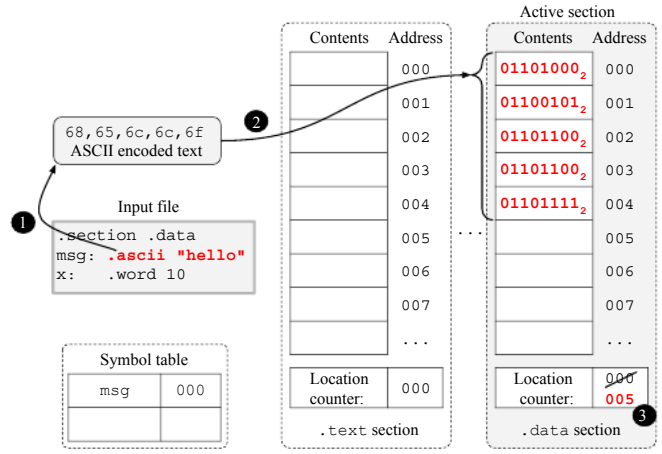


Figure 4.7: Processing the "`.ascii "hello"`" directive.

The next element in our assembly program is a label named "`x:`". In this case, the assembler registers the symbol named `x` at the symbol table and associates it the address that represents the current program location, *i.e.*, the address in the active location counter.

Finally, the last element in our assembly program is the `.word 10` directive, which instructs the assembler to add a 32-bit value to the active section. In this case, the assembler (1) encodes the 32-bit value as a sequence of four bytes, (2) stores the bytes on the active section using the little-endian convention[4], and (3) updates the location counter. Figure 4.8 illustrates this process.

4.7.2 The `.section` directive

As discussed in Section 3.3, assembly, object, and executable files are organized in sections. Also, by default, the GNU assembler tool adds information to the `.text`

[4]The little-endian convention specifies that the least significant byte must be associated with the smallest address - see Section 2.4.2 for more information on endianness.

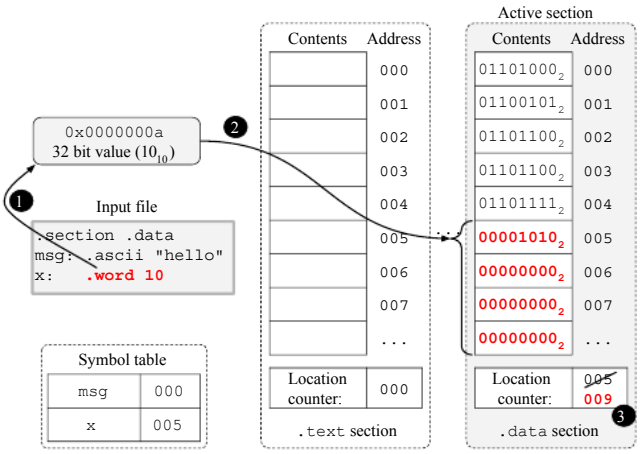


Figure 4.8: Processing the "`.word 10`" directive.

section. To instruct the assembler to add the assembled information into other sections, the programmer (or the compiler) may use the `.section secname` directive. As discussed in Section 4.6, when assembling a program, the assembler encodes and adds each assembly element to the active section. The "`.section secname`" changes the active section to `secname`, hence, all the information processed by the assembler after this directive is added to the `secname` section.

Program instructions are expected to be placed on the `.text` section, while constants, *i.e.*, read-only data, must be placed on the `.rodata` section. Also, initialized global variables must be placed on the `.data` section, and uninitialized global variables should be placed on the `.bss` section.

The following assembly code shows how the `.section` directive can be used to add the program instructions to the `.text` section and the program variables to the `.data` and `.rodata` sections.

```
1  .section .text
2  set_x:
3    la t1, x
4    sw a0, (t1)
5    ret
6  get_msg:
7    la a0, msg
8    ret
9  .section .data
10 x: .word 10
11 .section .rodata
12 msg: .string "Assembly rocks!"
```

NOTE: The RV32I GNU assembler also contains the "`.text`", "`.data`", and "`.bss`" directives, which are aliases to "`.section .text`", "`.section .data`", and the "`.section .bss`" directives, respectively.

4.7.3 Allocating variables on the `.bss` section

The `.bss` section is dedicated for storing uninitialized global variables. These variables need to be allocated on memory, but they do not need to be initialized by the loader when a program is executed. As a consequence, their initial value do not need to be stored on executable nor object files.

Since no information is stored on the `.bss` section in object and executable files, the GNU assembler does not allow assembly programs to add data to the `.bss` section. To illustrate this situation, let us consider the following code and assume it is stored on a file named `data-on-bss.s`:

```
1 .section .bss
2 x: .word 10
3 .section .text
4 set_x:
5   la t1, x
6   sw a0, (t1)
7   ret
```

This code is trying to use the `.word 10` directive to add a 32-bit value to the `.bss` section. However, when processing the `.word 10` directive, the GNU assembler stops assembling the code and emits the following error message:

```
1 $ riscv64-unknown-elf-as -march=rv32im data-on-bss.s -o data-on-bss.o
2 data-on-bss.s: Assembler messages:
3 data-on-bss.s:2: Error: attempt to store non-zero value in section `.bss'
```

To allocate variables on the `.bss` section it suffices to declare a label to identify the variable and advance the `.bss` location counter by the amount of bytes the variable require, so further variables are allocated on other address.

The `.skip N` directive is a directive that advances the location counter by `N` units and can be used to allocate space for variables on the `.bss` section. The following code shows how the `.skip` directive can be combined with labels to allocate space for three distinct variables: `x`, `V`, and `y`. In this example, the program is allocating 4 bytes for variables x and y and 80 bytes for variable V. As a consequence, labels `x`, `V`, and `y` will be associated with addresses `0x0`, `0x4`, and `0x54`, respectively.

```
1 .section .bss
2 x: .skip 4
3 V: .skip 80
4 y: .skip 4
```

NOTE: Some systems initialize the memory words dedicated to the `.bss` section with zeros when loading the program into the main memory for execution. Nonetheless, the programmer should not assume variables on the `.bss` section will be initialized with zeros.

4.7.4 The `.set` and `.equ` directives

The `.set name, expression` directive adds a symbol to the symbol table. It takes a name and an expression as arguments, evaluates an expression to a value and store the name and the resulting value into the symbol table. The following code shows how the `.set` directive can be used to associate values to symbols and use them on the program. In this case, the program first defines the symbol named `max_value` and associates it with value 42 (line 1). Then, it uses the `max_value` symbol on the `li` instruction (line 4).

```
1 .set max_value, 42
2
3 truncates_value_to_max:
```

```
4   li   t1, max_value
5   ble  a0, t1, ok
6   mv   a0, t1
7 ok:
8   ret
```

The `.equ` directive performs the same task as the `.set` directive.

4.7.5 The `.globl` directive

As discussed in Section 3.2.4, these program symbols can be classified as local or global symbols. By default, symbols automatically created for labels or explicitly created by the program using the `.set` or `.equ` directives are stored on the symbol table as local symbols. The `.globl` directive can be used to turn local symbols into global ones. The following code shows an example in which the `start` and the `max_temp` symbols are converted into global symbols by the `.globl` directive.

```
1 .globl max_value
2 .globl start
3
4 .set max_value, 42
5
6 start:
7   li  a0, max_value
8   jal process_temp
9   ret
```

4.7.6 The `.align` directive

Some Instruction Set Architectures require instructions or multi-byte data to be stored on addresses that are multiple of a given number. For example, the RV32I ISA requires instructions to be stored on addresses that are multiples of four.

The GNU assembler does not verify automatically whether or not RV32I instructions are assigned to addresses that are multiple of four. The following code, for example, is assembled without errors by the GNU assembler. In this case, the `j next` instruction is stored at addresses `0x0`, `0x1`, `0x2`, and `0x3`, the 8-bit value `0xa` is stored at address `0x4`, and the `ret` instruction is stored at addresses `0x5`, `0x6`, `0x7`, and `0x8`.

```
1 .text
2 foo:
3   j next
4   .byte 0xa
5 next:
6   ret
```

Even though the previous program was assembled by the assembler, a RV32I CPU will fail when trying to execute the `ret` instruction because it requires all instructions to be stored starting at addresses that are multiple of four.

The programmer (or the compiler) is responsible for keeping RV32I instructions aligned to 4-byte boundaries, *i.e.*, at addresses that are multiple of four. In the previous example, it could be done by advancing the location counter by three units right after adding the 8-bit value to the program (line 4). In this context, the following code would be executed correctly by a RV32I CPU.

```
1 .text
2 foo:
```

```
3   j next
4   .byte 0xa
5   .skip 3   # Advancing the location counter by 3 units.
6             # This is a very poor way of keeping the
7             # location counter aligned to a 4 byte boundary.
8 next:
9   ret
```

NOTE: Manually keeping track of the location counter, as in the previous example, is a cumbersome task and can lead to several problems. For example, trying to keep track and manually advance the location counter to ensure RV32I instructions are aligned to 4-byte boundaries may cause the assembler to generate invalid code without emitting warning nor error messages.

The proper way of ensuring the location counter is aligned is by using the `.align N` directive. The `.align N` directive checks if the location counter is a multiple of 2^N, if it is, it has no effect on the program, otherwise, it advances the location counter to the next value that is a multiple of 2^N.

The compiler usually inserts a `.align 2` directive[5] before routine labels to ensure the routine instructions start on addresses that are multiple of four. The following code shows an assembly code that uses the `.align 2` directive to align the location counter before each routine.

```
1 .text
2 .align 2
3 func1:
4   addi a0, a0, 2
5   ret
6 .align 2
7 func2:
8   addi a0, a0, 42
9   ret
```

Notice that, if the location counter already contains a value that is a multiple of 2^N, then the `.align N` directive has no effect on the location counter. Hence, since the code in the previous example starts at address zero and each assembly element is an assembly instruction that occupies exactly four bytes, the `.align 2` has no effect on the program.

The RV32I ISA allows programs to load and store data on unaligned memory addresses, however, for performance reasons, The RISC-V Instruction Set Manual [4] recommends 16-bit, 32-bit, and 64-bit values to be stored on addresses that are multiple of two, four, and eight, respectively. The following code shows who the `.align N` directive can be used to align multi-byte variables on the memory.

```
1 .data
2 .align 1
3 i: .half 1  # 16-bit variable initialized with value 1
4 .align 2
5 x: .word 9  # 32-bit variable initialized with value 9
6 .align 3
7 y: .dword 11 # 64-bit variable initialized with value 11
8 .bss
9 .align 3
10 z: .skip 8   # 64-bit variable (uninitialized)
```

[5] The `.align 2` directive aligns the location counter to a multiple of four (2^2).

Part II

User-level programming

Chapter 5

Introduction

Many computer systems are organized so that the software is divided into user and system software. The system software (*e.g.*, the operating system kernel and device drivers) is the software responsible for protecting and managing the whole system, including interacting with peripherals to perform input and output operations and loading and scheduling user applications for execution. The user software is usually limited to performing operations with data that is located on CPU registers and the main memory. Whenever the user software needs to perform a procedure that requires interacting with other parts of the system, such as reading data from a file or showing information on the computer display, it invokes the system software to perform the procedure on behalf of the user software.

In this part of the book, we will focus on the implementation of user software, *i.e.*, software that performs operations with data that is located on CPU registers and the main memory. We will also discuss how user software may invoke an operating system to perform other operations on its behalf, such as input/output from/to peripherals.

Part III covers system-level programming, including interacting with peripherals and securing the system against faulty or malicious user programs.

Chapter 6

The RV32I ISA

The RISC-V is a modular Instruction Set Architecture, allowing the design of a wide variety of microprocessors. This flexibility allows industry players to design microprocessors for applications with different requirements, including ultra-low-power and compact microprocessors for embedded devices and high-performing microprocessors for powerful servers running on data centers.

To achieve this flexibility, the RV32I Instruction Set Architecture relies on four base Instruction Set Architectures and several extensions that can be combined with the base Instruction Set Architectures to implement specialized versions of the Instruction Set Architecture. Table 6.1 presents the base Instruction Set Architectures and some of its extensions[1].

Base ISAs

Name	Description
RV32I	32-bit integer instruction set
RV32E	32-bit integer instruction set for embedded microprocessors
RV64I	64-bit integer instruction set
RV128I	128-bit integer instruction set

Extensions

Suffix	Description
M	Standard extension for integer multiplication and division
A	Standard extension for atomic instructions
F	Standard extension for single-precision Floating-Point
D	Standard extension for double-precision Floating-Point
G	Shorthand for the base and above extensions
Q	Standard extension for quad-precision floating-point
L	Standard extension for decimal floating-point
C	Standard extension for compressed instructions
B	Standard extension for bit manipulation
J	Standard extension for dynamically translated languages
T	Standard extension for transactional memory
P	Standard extension for packed-SIMD instructions
V	Standard extension for vector operations
N	Standard extension for user-level interrupts
H	Standard extension for hypervisor

Table 6.1: RISC-V base ISAs and extensions.

In this book, we will focus on the RV32IM , which includes the RV32I base and extension M, which includes instructions for integer multiplication and division. The RV32IM has the following properties:

[1]Some of these extensions are still under development. Refer to the official RISC-V consortium (`http://www.risc-v.com`) for up-to-date information on current base ISAs and extensions

- It supports 32-bit address spaces;
- It contains thirty-three 32-bit registers;
- It represents signed integer values in two's-complement;
- It contains the basic instructions, including integer computational instructions, integer loads, integer stores, and control-flow instructions; and
- It also contains instructions to multiply and divide values held in the integer registers (M extension).

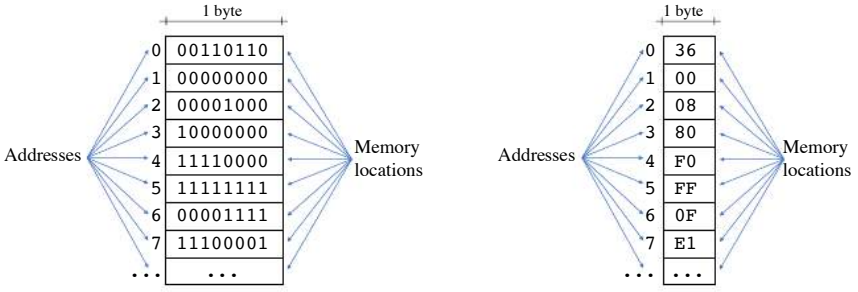
6.1 Datatypes and memory organization

ISA native datatypes are datatypes that can be naturally processed by the ISA. Table 6.2 shows the RV32I native datatypes and their respective sizes in bytes.

RV32I native datatype name	size in bytes
`byte`	1
`unsigned byte`	1
`halfword`	2
`unsigned halfword`	2
`word`	4
`unsigned word`	4

Table 6.2: RV32I native datatypes

Similar to other modern s, RISC-V is designed to work with byte addressable memories, *i.e.*, memories in which each memory location stores a single byte and is associated with a unique address, as illustrated in Figure 6.1.



(a) (b)

Figure 6.1: Organization of a byte addressable memory with its contents represented in the binary (a) and the hexadecimal (b) bases.

Datatypes larger than one byte are stored on multiple memory locations. Hence, when storing a `halfword` (`word`) datatype value on memory, the stores the two (four) bytes on two (four) consecutive memory locations.

When translating a program written in "C" to RV32I assembly code, the **C datatypes** must be converted into RISC-V native datatypes. Table 6.3 shows the mappings from C native datatypes to RV32I native datatypes[2]. All pointers in C (*e.g.*, `int*`, `char*`, and `void*`) represent memory addresses and are mapped to the `unsigned word` datatype.

[2]This mapping is valid for the RISC-V ilp32 ABI, which is discussed in this book and further discussed in Chapter 8.

C datatype	RV32I native datatype	size in bytes
`bool`	`byte`	1
`char`	`byte`	1
`unsigned char`	`unsigned byte`	1
`short`	`halfword`	2
`unsigned short`	`unsigned halfword`	2
`int`	`word`	4
`unsigned int`	`unsigned word`	4
`long`	`word`	4
`unsigned long`	`unsigned word`	4
`void*`	`unsigned word`	4

Table 6.3: Mapping C datatypes to RV32I native datatypes

6.2 RV32I registers

The RV32I unprivileged ISA contains thirty three 32-bit registers, also known as unprivileged registers.

Register `x0` is a special register that is hard-wired to zero, *i.e.*, it always returns the value zero when read[3]. Register `pc` holds the **program counter**, *i.e.*, the address of the next instruction to be executed. Its contents are automatically updated every time an instruction is executed and may be updated by special instructions, called control-flow instructions[4].

The remaining registers (`x1-x31`) are general purpose registers and can be used interchangeably. Nonetheless, as we will discuss later, it is usually important to follow a register usage standard. For example, always using the same set of registers to pass parameters when invoking a function. To facilitate the programming, these registers are given **aliases**, that can be used when writing assembly code. The goal is to allow the programmer to use more meaningful register names when programming. For example, writing `a0`, instead of `x10`, to refer to the register that is holding the first **a**rgument to a function. Table 6.4 shows the list of unprivileged registers, their aliases and the descriptions of their aliases.

Register	Alias	Description	Caller-save	Callee-save
`x0`	`zero`	Hard-wired zero		
`x1`	`ra`	Return address	x	
`x2`	`sp`	Stack pointer		x
`x3`	`gp`	Global pointer		
`x4`	`tp`	Thread pointer		
`x5-x7`	`t0-t2`	Temporaries 0 to 2	x	
`x8`	`s0/fp`	Saved register 0 / Frame pointer		x
`x9`	`s1`	Saved register 1		x
`x10-17`	`a0-a7`	Function arguments 0 to 7	x	
`x18-27`	`s2-s11`	Saved registers 2 to 11		x
`x28-31`	`t3-t6`	Temporaries 3 to 6	x	
`pc`	`pc`	Program counter		

Table 6.4: RV32I unprivileged registers

6.3 Load/Store architecture

A **Load/Store architecture** is an instruction set architecture that requires values to be loaded/stored explicitly from/to memory before operating on them. In other

[3]Values written to this register are discarded.
[4]These instructions will be discussed on Section 6.7.

words, to read/write a value from/to memory, the software must execute a load/store instruction.

The RISC-V is a Load/Store architecture, hence, to perform operations (*e.g.*, arithmetic operations) on data stored on memory, it requires the data to be first retrieved from memory into a register by executing a load instruction. As an example, let us consider the following assembly code, which loads a value from memory, multiply it by two, and stores the result on memory.

```
1 lw  a5, 0(a0)
2 add a6, a5, a5
3 sw  a6, 0(a0)
```

The first instruction, called load word and indicated by the mnemonic `lw`, is a load instruction. It retrieves a `word` value from memory and stores it on register `a5`. The expression `0(a0)` indicates the address of the memory position that stores the value that must be loaded. In this case, the address is the sum of the contents of register `a0` and the constant `0`. In other words, in case register `a0` contains the value `8000` when this load instruction is executed, the hardware will retrieve the data from the memory location associated with address `8000`.

The second instruction, indicated by the mnemonic `add`, adds two values and stores the result on a register. In this case, it is adding the values from registers `a5` and `a5` and storing the result on register `a6`. Notice that, since both source operands are the same, *i.e.*, `a5`, the result is equivalent to multiplying the contents of `a5` by two.

Finally, the third instruction, called store word and indicated by the mnemonic `sw`, stores the value from register `a6` into memory. Again, the expression `0(a0)` indicates the address of the memory position that will receive the data.

6.4 Pseudo-instructions

When assembling an assembly program, the assembler converts each assembly instruction (encoded as plain text) into its corresponding machine instruction (encoded as binary). For example, the assembly instruction "`add x10, x11, x12`" is converted into its corresponding machine instruction, which is encoded in four bytes as "`0x00c58533`".

A **pseudo-instruction** is an assembly instruction that does not have a corresponding machine instruction on the , but can be translated automatically by the assembler into one or more alternative machine instructions to achieve the same effect. As an example, the no operation instruction, or "`nop`", is a RV32I pseudo-instruction that is converted by the assembler into the "`addi x0, x0, 0`" instruction[5]. Another example is the "`mv`" instruction, which copies the contents of one register into another. In this case, the pseudo-instruction "`mv a5, a7`", which copies the contents of `a7` into `a5`, is converted into the instruction "`addi a5, a7, 0`", which adds zero to the value in `a7` and stores the result on register `a5`.

Since the focus of the book is on assembly programming, the remaining of this book will not differentiate pseudo-instructions from real RV32I machine instructions. We refer the reader to the RISC-V Instruction Set Manual [4] for a full list of real RV32I machine instructions and pseudo-instructions.

6.5 Logic, shift, and arithmetic instructions

Logic, shift, and arithmetic instructions are instructions that perform logic, shift, and arithmetic operations on data.

[5]This instruction adds zero to zero and discards the results by storing it on register x0, which is hard-wired to zero.

6.5.1 Instructions syntax and operands

All RV32I logic, shift, and arithmetic instructions operate on data indicated by the instruction operands. These instructions contain three operands, one target operand and two source operands. The first operand indicates the target register (`rd`), *i.e.*, the register in which the result of the operation will be stored. The second operand indicates a register (`rs1`) that contains the first source operand. The third operand indicates the second source operand, which may be another register (`rs2`) or an immediate value[6] (`imm`). The syntax of logic, shift and arithmetic instructions can be either:

```
MNM rd, rs1, rs2
```

or

```
MNM rd, rs1, imm
```

where `MNM` indicates the instruction mnemonic, `rd` indicates the target register, `rs1` indicates the first source operand and `rs2` (or `imm`) indicates the second operand. The following assembly code shows examples of logic, shift and arithmetic RV32I assembly instructions:

```
1 and  a0, a2, a6  # a0 <= a2 & a6
2 slli a1, a3, 2   # a1 <= a3 << 2
3 sub  a4, a5, a6  # a4 <= a5 - a6
```

The first instruction performs a bitwise "and" operation using the values from `a2` and `a6` and stores the result on `a0`. The second instruction shifts the value from `a3` to the left twice and stores the result on `a1`. In this case, the second source operand (`2`) is an immediate value (`imm`). Finally, the third instruction subtracts the value at `a6` from the value at `a5` and stores the result on register `a4`.

Any general purpose register (`x0-x31`) may be used as `rd`, `rs1`, or `rs2`. However, it is worth noticing that if `x0` (`zero`) is indicated as a target operand (`rd`), then the result will be discarded. This happens because `x0` is hard-wired to zero.

6.5.2 Dealing with large immediate values

The immediate value is a constant encoded into the instruction itself. Besides this value, the instruction must also encode other information, such as the opcode and the other operands. Since all RV32I instructions have exactly 32-bits, the amount of bits available to encode the immediate value is smaller than 32 bits. In fact, the RV32I arithmetic, logic and shift instructions can only encode immediate values that can be represented as a 12-bit twos'-complement signed number. In other words, **the immediate values used as operands on these instructions must be greater or equal to -2048 (-2^{11}) and less or equal to 2047 ($2^{11}-1$)**.

The following code shows a set of invalid assembly instructions:

```
1 add a0, a5, 2048
2 add a0, a5, 10000
3 add a0, a5, -3000
```

They are invalid because the assembler cannot encode the immediate values into the instruction (notice that they may not be encoded as 12-bit twos'-complement signed numbers). In this example, the assembler will fail to assemble the code and potentially show an error message. The GNU assembler (`as`) shows the following message when trying to assemble an assembly program with these instructions:

[6] An immediate value is a constant value encoded into the instruction itself.

```
prog.s: Assembler messages:
prog.s:1: Error: illegal operands `add a0,a5,2048'
prog.s:2: Error: illegal operands `add a0,a5,10000'
prog.s:3: Error: illegal operands `add a0,a5,-3000'
```

To perform operations with immediate values less than -2048 or greater than 2047, the programmer could employ multiple instructions to compose the value, store it into a register, and use an instruction that reads the second source operand from a register. There are several ways of composing these values using RV32I instructions. As an example, one could load a small constant (*e.g.*, 1000) into a register, shift its value to the left to multiply it by a power of two, and add another small constant to produce the desired value. The following assembly code produces the value 4005 by loading the value 1000 into `a5`, shifting it twice to the left[7], and adding 5 to it.

```
ADD  a5, x0, 1000
SLLI a5, a5, 2
ADD  a5, a5, 5
```

In RISC-V, the **recommended way of loading immediate values into registers is by using the "load immediate" pseudo-instruction, or** `li`. This pseudo-instruction is automatically converted by the assembler to the best sequence of machine instructions to compose the desired value. The syntax of the load immediate instruction is:

```
li rd, imm
```

where `rd` indicates the target register and `imm` is the desired immediate value.

6.5.3 Logic instructions

Table 6.5 shows the RV32I logic instructions. The `and/or/xor` instruction performs the bitwise "and"/"or"/"xor" operation on values stored in registers `rs1` and `rs2` and stores the result on register `rd`. The `andi/ori/xori` perform the operation using the value stored in register `rs1` and an immediate value.

Instruction	Description
`and rd, rs1, rs2`	Performs the bitwise "and" operation on `rs1` and `rs2` and stores the result on `rd`.
`or rd, rs1, rs2`	Performs the bitwise "or" operation on `rs1` and `rs2` and stores the result on `rd`.
`xor rd, rs1, rs2`	Performs the bitwise "xor" operation on `rs1` and `rs2` and stores the result on `rd`.
`andi rd, rs1, imm`	Performs the bitwise "and" operation on `rs1` and `imm` and stores the result on `rd`.
`ori rd, rs1, imm`	Performs the bitwise "or" operation on `rs1` and `imm` and stores the result on `rd`.
`xori rd, rs1, imm`	Performs the bitwise "xor" operation on `rs1` and `imm` and stores the result on `rd`.

Table 6.5: RV32I logic instructions

The following assembly code shows examples of valid logic instructions:

```
and a0, a2, s2  # a0 <= a2 & s2
or  a1, a3, s2  # a1 <= a3 | s2
xor a2, a2, a1  # a2 <= a2 ^ a1
```

[7] shifting a binary-encoded value N times to the left is equivalent to multiplying it by 2^N

```
4 andi a0, a2, 3  # a0 <= a2 & 3
5 ori  a1, a3, 4  # a1 <= a3 | 4
6 xori a2, a2, 1  # a2 <= a2 ^ 1
```

The following code loads two immediate values into registers `a1` and `a2` and performs an "and" operation. The result of this operation (`0x0000AB00`) is stored in register `a0`.

```
1 li  a1, 0xFE01AB23  # a1 <= 0xFE01AB23
2 li  a2, 0x0000FF00  # a2 <= 0x0000FF00
3 and a0, a1, a2      # a0 <= a1 & a2
```

6.5.4 Shift instructions

Shift instructions are used to shift binary values left or right. These instructions may be used to pack or unpack bits into `words` or to perform arithmetic multiply and divide operations. Table 6.6 shows the RV32I shift instructions.

Instruction	Description
`sll rd, rs1, rs2`	Performs a logical left shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the value on `rs2`.
`srl rd, rs1, rs2`	Performs a logical right shift on the value at `rs1` and stores the result on `rd`. The amount of right shifts is indicated by the value on `rs2`.
`sra rd, rs1, rs2`	Performs an arithmetic right shift on the value at `rs1` and stores the result on `rd`. The amount of right shifts is indicated by the value on `rs2`.
`slli rd, rs1, imm`	Performs a logical left shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.
`srli rd, rs1, imm`	Performs a logical right shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.
`srai rd, rs1, imm`	Performs an arithmetic right shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.

Table 6.6: RV32I shift instructions

The **logical left shift instructions** (`sll` or `slli`) perform a logical left shift on a value that is stored on a register. The amount of shifts is indicated as an operand to the instruction and can be either the value in a register or an immediate value. The following code shows examples of logical left shift instructions. The first shift instruction (`slli`) shifts the value in `a2` twice to the left and stores the result in `a0`. The second one (`sll`) performs a similar operation, but the amount of shifts to the left is defined by the value in `a3`.

```
1 li   a2, 24      # a2 <= 24
2 slli a0, a2, 2   # a0 <= a2 << 2
3 sll  a1, a2, a3  # a0 <= a2 << a3
```

The first two instructions on the previous code load the immediate value 24 into register `a2`, shift its contents twice to the left and stores the result on `a0`. The immediate value 24 is represented by the binary number

```
1 00000000 00000000 00000000 00011000
```

The logical left shift operation shifts the bits to the left, discarding the leftmost bits and adding zeros to the right. Hence, after shifting it twice to the left, the result will be the binary number

```
00000000 00000000 00000000 01100000
```

This binary number corresponds to the decimal value 96 and is equivalent to 24×4. In fact, **logical left shift operations may be used to multiply numbers by powers of two**. In this context, **shifting a value N times to the left is equivalent to multiplying the value by** 2^N. The following assembly code shows examples of logical left shift instructions being used to multiply the value in register `a3` by 2, 4, and 8, respectively.

```
slli a0, a3, 1   # a0 <= a2 * 2
slli a1, a3, 2   # a0 <= a2 * 4
slli a2, a3, 3   # a0 <= a2 * 8
```

Shift operations are easier to implement in hardware than the multiply operation and usually take less time and/or energy to be executed. As a consequence, whenever possible, compilers try to generate these instructions to perform multiplications.

The **logical right shift instructions** (`srl` or `srli`) performs a logical right shift on a value that is stored on a register. Similar to the logical left shift instructions, the amount of shifts is indicated as an operand to the instruction and can be either the value in a register or an immediate value. The following code shows examples of logical right shift instructions. The first shift instruction (`slri`) shifts the value in `a5` twice to the right and stores the result in `a0`. The second one (`sll`) performs a similar operation, but the amount of shifts to the right is defined by the value in `a7`.

```
li   a5, 24      # a5 <= 24
srli a0, a5, 2   # a0 <= a5 >> 2
srl  a1, a5, a7  # a0 <= a5 >> a7
```

The first two instructions on the previous code load the immediate value 24 into register `a5`, shift its contents twice to the right and stores the result on `a0`. The immediate value 24 is represented by the binary number

```
00000000 00000000 00000000 00011000
```

The logical right shift operation shifts the bits to the right, discarding the rightmost bits and adding zeros to the left. Hence, after shifting it twice to the right, the result will be the binary number

```
00000000 00000000 00000000 00000110
```

This binary number corresponds to the decimal value 6 and is equivalent to 24/4. In fact, logical right shift operations may be used to integer divide numbers by powers of two.

In the previous example, we verified that by performing a logical right shift operation twice on value 24 resulted in value 6, *i.e.*, 24/4. Nonetheless, this is not valid for negative numbers. Let us take the value -24 as an example. This value is represented by the following binary number in RISC-V[8]

```
11111111 11111111 11111111 11101000
```

[8]The RV32I Instruction Set Architecture uses the two's-complement representation for signed numbers.

The logical right shift operation shifts the bits to the right, discarding the rightmost bits and adding zeros to the left. Hence, after shifting it twice to the right, the result will be the binary number

```
00111111 11111111 11111111 11111010
```

Notice, however, that this number is not negative and does not correspond to the division of −24 by four. In fact, it is a very large positive number (1 073 741 818).

In case we assume the unsigned representation, the previous binary number represents the value 4 294 967 272, instead of −24. In this case, the result of applying the logical right shift operation twice would result in the division of this number by four, *i.e.*, 1 073 741 818.

In summary, the **logical right shift operation may only be used to divide unsigned numbers**. In this case, **shifting an unsigned number N times to the right with a logical right shift operation is equivalent to integer dividing the unsigned number by** 2^N.

The **arithmetic right shift** instructions (`sra` or `srai`) perform an arithmetic right shift on a value that is stored on a register. Similar to the logical right shift instructions, the amount of shifts to the right is indicated as an operand on the instruction and can be either the value in a register or an immediate value. The following code shows examples of arithmetic right shift instructions. The first arithmetic right shift instruction (`srai`) shifts the value in `a5` twice to the right and stores the result in `a0`. The second one (`sra`) performs a similar operation, but the amount of shifts to the right is defined by the value in `a7`.

```
1 li   a5, -24     # a5 <= -24
2 srai a0, a5, 2   # a0 <= a5 >> 2
3 sra  a1, a5, a7  # a0 <= a5 >> a7
```

The first two instructions on the previous code load the immediate value −24 into register `a5`, shift its contents twice to the right and stores the result on `a0`. As discussed before, the immediate value −24 is represented by the binary number

```
11111111 11111111 11111111 11101000
```

The arithmetic right shift operation shifts the bits to the right, discarding the rightmost bits. Instead of simply adding zeros to the left, it replicates the leftmost bit, *i.e.*, if the leftmost bit is equal to `1`, then it inserts ones on the left. In case the leftmost bit is equal to `0`, then it inserts zeros on the left. As a result, after performing an arithmetic right shift twice on the previous, the result will be the binary number

```
11111111 11111111 11111111 11111010
```

This binary number corresponds to the decimal value −6 and is equivalent to −24/4. In fact, arithmetic right shift operations may be used to integer divide signed numbers by powers of two. Notice that this instruction can also be used to integer divide positive signed numbers by powers of two. It works because the leftmost bit of positive signed numbers is zero. Hence, the arithmetic right shift operation will push zeros to the left when shifting the value.

In summary, the **arithmetic right shift operation may only be used to integer divide signed numbers**. In this case, **shifting a signed number N times to the right with an arithmetic right shift operation is equivalent to integer dividing the signed number by** 2^N.

6.5.5 Arithmetic instructions

Table 6.7 shows the RV32I arithmetic instructions and instructions from the M extension (`mul`, `div`, and `rem`).

Instruction	Description
`add rd, rs1, rs2`	Adds the values in `rs1` and `rs2` and stores the result on `rd`.
`sub rd, rs1, rs2`	Subtracts the value in `rs2` from the value in `rs1` and stores the result on `rd`.
`addi rd, rs1, imm`	Adds the value in `rs1` to the immediate value `imm` and stores the result on `rd`.
`mul rd, rs1, rs2`	Multiplies the values in `rs1` and `rs2` and stores the result on `rd`.
`div{u} rd, rs1, rs2`	Divides the value in `rs1` by the value in `rs2` and stores the result on `rd`. The `U` suffix is optional and must be used to indicate that the values in `rs1` and `rs2` are unsigned.
`rem{u} rd, rs1, rs2`	Calculates the remainder of the division of the value in `rs1` by the value in `rs2` and stores the result on `rd`. The `U` suffix is optional and must be used to indicate that the values in `rs1` and `rs2` are unsigned.

Table 6.7: RV32I arithmetic instructions

The **add instructions** (`add` and `addi`) adds two numbers and stores the result on a register (`rd`). In both cases, the first number is retrieved from the register `rs1`. Instruction `add` retrieves the second number from register `rs2` while instruction `addi` uses the immediate value `imm`.

The **subtract instruction** (`sub`) subtracts the value in `rs2` from the value in `rs1` and stores the result on `rd`. The RV32I does not contain a `subi` instruction, *i.e.*, an instruction that subtracts an immediate values from the contents of a register and stores the result on another register. Nonetheless, it is worth noticing that a programmer can easily achieve this effect by adding a negative immediate value using the `addi` instruction. The following code shows an example of an instruction that subtracts the immediate value 10 from the contents of register `a2` and stores the result on `a0` using an `addi` instruction.

```
1 addi  a0, a2, -10 # a0 <= a2 - 10
```

The **multiply instruction** (`mul`) multiplies the values in `rs1` and `rs2` and stores the result on `rd`.

The **divide instructions** (`div` and `divu`) divide the value in `rs1` by the value in `rs2` and stores the result on `rd`. Instruction `div` divides signed numbers while `divu` divides unsigned numbers.

The **remainder instructions** (`rem` and `remu`) computes the remainder of the division of the value in `rs1` by the value in `rs2` and stores the result on `rd`. Instruction `rem` computes the remainder for divisions of signed numbers while `remu` computes the remainder for divisions of unsigned numbers.

The following assembly code shows examples of RV32IM arithmetic instructions:

```
1 add   a0, a2, t2  # a0 <= a2 + t2
2 addi  a0, a2, 10  # a0 <= a2 + 10
3 sub   a1, t3, a0  # a1 <= t3 - a0
4 mul   a0, a1, a2  # a0 <= a1 * a2
5 div   a1, a3, a5  # a1 <= a3 / a5
6 rem   a1, a3, a5  # a1 <= a3 % a5
7 remu  a1, a3, a5  # a1 <= a3 % a5
```

Did you know? In case you are programming for an RV32I that lacks the M extension, *i.e.*, it does not contain the multiply and divide instructions, you may be able to combine arithmetic and shift instructions to perform multiplications and

divisions. The following assembly code shows an example of how the `slli` and `addi` instructions may be used to multiply the value of `a2` by 5 and by 10:

```
1 slli  a0, a2,  2  # a0 <= a2 * 4
2 add   a0, a0, a2  # a0 <= a0 + a2, i.e., a2 * 5
3 slli  a1, a0,  1  # a1 <= a0 * 2, i.e., a2 * 10
```

6.6 Data movement instructions

RV32I data movement instructions can be used to load data from memory into registers, store register data into memory, copy data from one register to another, or load immediate values or label addresses into registers. Table 6.8 shows the RV32I data movement instructions and Table 6.9 shows the RV32I data movement pseudo-instructions.

Instruction	Description
`lw rd, imm(rs1)`	Loads a 32-bit `signed` or `unsigned word` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lh rd, imm(rs1)`	Loads a 16-bit `signed halfword` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lhu rd, imm(rs1)`	Loads a 16-bit `unsigned halfword` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lb rd, imm(rs1)`	Loads a 8-bit `signed byte` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lbu rd, imm(rs1)`	Loads a 8-bit `unsigned byte` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`sw rs1, imm(rs2)`	Stores the 32-bit value at register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.
`sh rs1, imm(rs2)`	Stores the 16 least significant bits from register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.
`sb rs1, imm(rs2)`	Stores the 8 least significant bits from register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.

Table 6.8: RV32I data movement instructions

6.6.1 Load instructions

All RV32I load instructions (`lw`, `lh`, `lhu`, `lb`, and `lbu`) load values from memory into a register. The assembly syntax for these instructions is:

```
MNM rd, imm(rs1)
```

where `MNM` indicates the instruction mnemonic. The first operand (`rd`) indicates the target register, *i.e.*, the one that will receive the value loaded from memory. The second (`imm`) and third (`rs1`) operands indicate an immediate value and a register, respectively. The values of these two operands are added together to calculate the memory address.

The **load word** instruction (`lw`) loads a 32-bit `word` from memory into a register. Since a `word` datatype has four bytes, this instruction loads four bytes from four

Instruction	Description
`mv rd, rs`	Copies the value from register `rs` into register `rd`.
`li rd, imm`	Loads the immediate value `imm` into register `rd`.
`la rd, rot`	Loads the label address `rot` into register `rd`.
`L{W|H|HU|B|BU} rd, lab`	For each one of the `lw`, `lh`, `lhu`, `lb`, and `lbu` machine instructions there is a pseudo-instruction that performs the same operation, but the memory address is calculated based on a label (`lab`).
`S{W|H|B} rd, lab`	For each one of the `sw`, `sh`, and `sb` machine instructions there is a pseudo-instruction that performs the same operation, but the memory address is calculated based on a label (`lab`).

Table 6.9: RV32I data movement pseudo-instructions

consecutive memory positions (starting at the calculated address) and stores these four bytes into the target register. The RV32I follows the little-endian endianness format, hence, the byte loaded from the memory position associated with the smallest address is loaded into the register's least significant byte. Figure 6.2 illustrates a value being loaded from memory into register `a0`. In this example, the data (a four-byte `word`) is being loaded from four consecutive memory locations, starting at address `8000`. The start address is calculated by adding the immediate value (`0`) to the contents of register `a2` (8000_{10}).

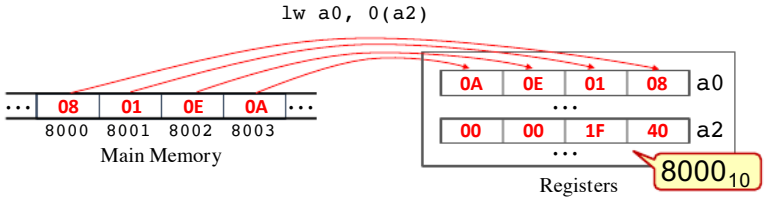


Figure 6.2: Example of a `word` (`0x0A0E0108`) being loaded by a load word instruction

The load word instruction is used to load `int`, `unsigned int`, `long`, `unsigned long`, and pointers from memory[9].

The **load unsigned byte** instruction (`lbu`) loads a 8-bit `unsigned byte` from memory into a register. Since registers have 32 bits, or four bytes, the `unsigned byte` loaded from memory is stored on the least significant register byte and the other three register bytes are set to zero. Figure 6.3 illustrates an `unsigned byte` value being loaded from memory into register `a0`. In this example, the data (an `unsigned byte`) is being loaded from the memory location associated with address `8000`, which is calculated by adding the immediate value (`0`) to the contents of register `a2` (8000_{10}).

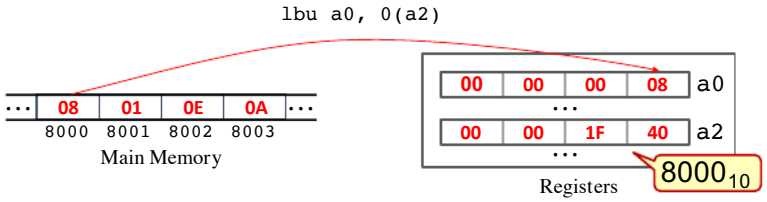


Figure 6.3: Example of `unsigned byte` (`0x08`) being loaded by a load unsigned byte instruction

The load unsigned byte instruction is used to load `unsigned char` datatype values

[9]Section 6.1 discusses the mappings from C datatypes to the RV32I ISA native datatypes.

from memory.

The **load byte** instruction (`lb`) loads a 8-bit `signed byte` from memory into a register. Again, since registers have 32 bits, the `signed byte` loaded from memory is stored on the least significant register byte. In case the value is non-negative, the other three register bytes are set to zero. In case it is negative, the bits of the other three register bytes are set to one. Figure 6.4 illustrates a non-negative `signed byte` value (`0x08` = 8_{10}) being loaded from memory into register `a0`. In this example, the data, a non-negative `unsigned byte`, is being loaded from the memory location associated with address `8000`, which is calculated by adding the immediate value (`0`) to the contents of register `a2` (8000_{10}). Notice that the three most significant register bytes are set to zero.

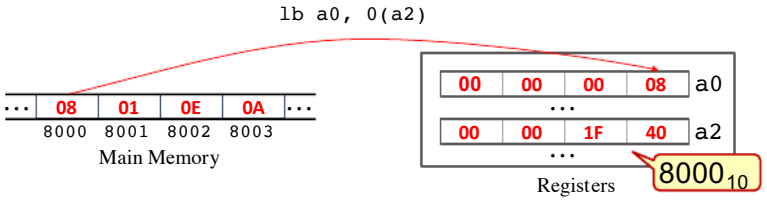


Figure 6.4: Example of a non-negative `signed byte` value (`0x08` = 8_{10}) being loaded by the load byte instruction

Figure 6.5 illustrates a negative `signed byte` value (`0xFE` = -2_{10}) being loaded from memory into register `a0`. Again, the data is being loaded from the memory location associated with address `8000`, which is calculated by adding the immediate value (`0`) to the contents of register `a2` (8000_{10}). Notice, however, that the bits of the three most significant register bytes are set to ones and the final value is properly set to `0xFFFFFFFE`, *i.e.*, -2_{10}.

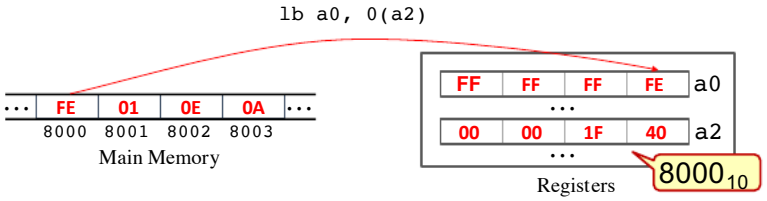


Figure 6.5: Example of a negative `signed byte` value (`0xFE` = -2_{10}) being loaded by the load byte instruction

The load byte instruction is used to load `char` C datatype values from memory.

The **load unsigned halfword** instruction (`lhu`) loads a 16-bit `unsigned halfword` from memory into a register. Since an `unsigned halfword` datatype has two bytes, this instruction loads two bytes from two consecutive memory positions (starting at the calculated address) and stores these two bytes into the target register. Again, since the RV32I follows the little-endian endianness format, the byte loaded from the memory position associated with the smallest address is loaded into the register's least significant byte and the second byte loaded into the register's second-least significant byte. The other two register bytes are set to zero. Figure 6.6 illustrates an `unsigned halfword` value (`0x0108`) being loaded from memory into register `a0`. In this example, the data, an `unsigned halfword`, is being loaded from the memory locations starting at address `8000`, which is calculated by adding the immediate value (`0`) to the contents of register `a2` (8000_{10}). Notice that the two most-significant register bytes are set to zero.

The load unsigned halfword instruction is used to load `unsigned short` C datatype values from memory.

The **load halfword** instruction (`lh`) loads a 16-bit signed `halfword` from memory into a register. Since a `halfword` datatype has two bytes, this instruction loads two

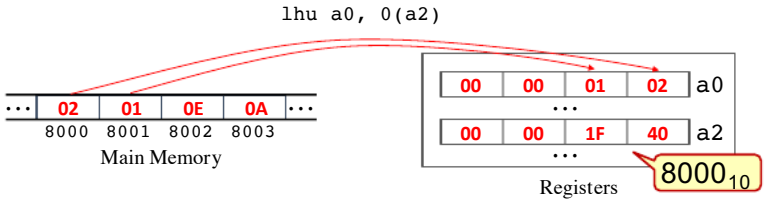


Figure 6.6: Example of an `unsigned halfword` value (`0x0108`) being loaded by the load unsigned halfword instruction

bytes from two consecutive memory positions (starting at the calculated address) and stores these two bytes into the target register. Again, since the RV32I follows the little-endian endianness format, the byte loaded from the memory position associated with the smallest address is loaded into the register's least significant byte and the second byte loaded into the register's second-least significant byte. In case the value is non-negative, the bits of the other two bytes of the register are set to zeros, and in case it is negative, they are set to ones. Figure 6.7 illustrates a non-negative `halfword` value (`0x0102` = 258_{10}) being loaded from memory into register `a0`. In this example, the data, a non-negative `halfword`, is being loaded from the memory location associated with address `8000`, which is calculated by adding the immediate value (`0`) to the contents of register `a2` (8000_{10}). Notice that the two most significant register bytes are properly set to zero and the final result is `0x00000102`, *i.e.*, 258_{10}.

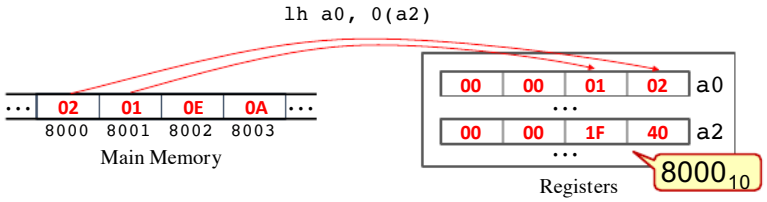


Figure 6.7: Example of a non-negative `signed halfword` value (`0x0102` = 258_{10}) being loaded by the load halfword instruction

Figure 6.8 illustrates a negative `halfword` value (`0xFFFE` = -2_{10}) being loaded from memory into register `a0`. Again, the data is being loaded from the memory locations starting at address `8000`, which is calculated by adding the immediate value (`0`) to the contents of register `a2` (8000_{10}). Notice, however, that the bits of the two most significant register bytes are set to ones and the final value is properly set to `0xFFFFFFFE`, *i.e.*, -2_{10}.

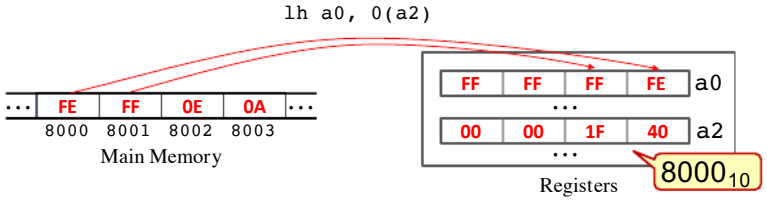


Figure 6.8: Example of a negative `signed halfword` value (`0xFFFE` = -2_{10}) being loaded by the load halfword instruction

The load halfword instruction is used to load `short` C datatype values from memory.

6.6.2 Store instructions

All RV32I store instructions (`sw`, `sh`, and `sb`) store values from registers into memory. The assembly syntax for these instructions is:

```
MNM rs1, imm(rs2)
```

where `MNM` indicates the instruction mnemonic. The first operand (`rs1`) indicates the source register, *i.e.*, the register that contains the value to be stored on memory. The second (`imm`) and third (`rs2`) operands indicate an immediate value and a register, respectively. The value of these two operands are added together to calculate the memory address.

The **store word** instruction (`sw`) stores a 32-bit `word` from `rs1` into the memory. Since a `word` datatype has four bytes, this instruction stores the four bytes into four consecutive memory positions (starting at the calculated address). The RV32I follows the little-endian endianness format, hence, the least significant byte is stored on the memory position associated with the smallest address and so on. Figure 6.9 illustrates a value from `a0` being stored into the main memory by a `sw` instruction. In this example, the data (a four-byte `word`) is being stored on four consecutive memory locations, starting at address `8000`. The start address is calculated by adding the immediate value (`0`) to the contents of register `a2` (8000_{10}).

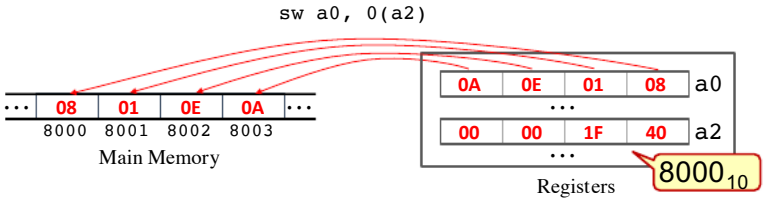


Figure 6.9: Example of a `word` value (`0x0A0E0108`) being stored by the store word instruction

The store word instruction is used to store `int`, `unsigned int`, `long`, `unsigned long`, and pointers into memory.

The **store halfword** instruction (`sh`) stores the least significant 16-bit `halfword` from `rs1` into memory. Since a `halfword` datatype has two bytes, this instruction stores the two least significant bytes from register `rs1` into two consecutive memory positions (starting at the calculated address). The RV32I follows the little-endian endianness format, hence, the least significant byte is stored on the memory position associated with the smallest address and so on. Figure 6.10 illustrates a `halfword` value from `a0` being stored into the main memory by a `sh` instruction. In this example, the data (a two-byte `halfword`) is being stored on two consecutive memory locations, starting at address `8000`. The start address is calculated by adding the immediate value (`0`) to the contents of register `a2` (8000_{10}).

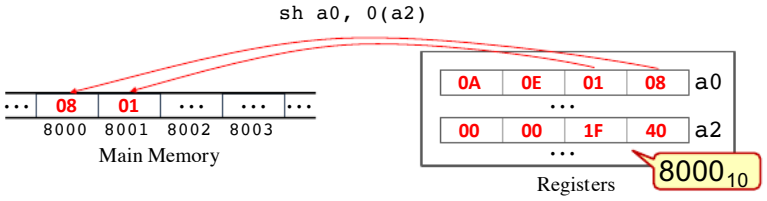


Figure 6.10: Example of a `halfword` value (`0x0108`) being stored by the store halfword instruction

The store halfword instruction is used to store `short` and `unsigned short` C datatype values into memory.

The **store byte** instruction (`sb`) stores the least significant 8-bit `byte` from `rs1` into memory. Since a `byte` datatype has only one byte, this instruction stores the

least significant byte from register `rs1` into a single memory position, indicated by the calculated address. Figure 6.11 illustrates a `byte` value from `a0` being stored into the main memory by a `sb` instruction. In this example, the data (a single `byte`) is being stored on a single memory location, at address `8000`. The memory address is calculated by adding the immediate value (`0`) to the contents of register `a2` (8000_{10}).

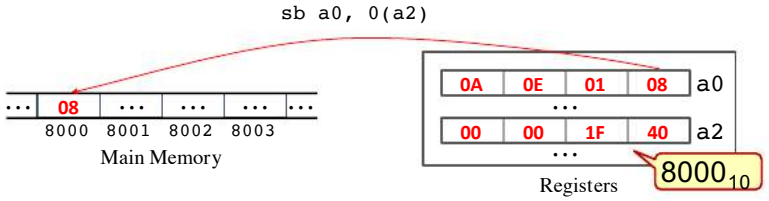


Figure 6.11: Example of a `byte` value (`0x08`) being stored by the store byte instruction

The store byte instruction is used to store `char` and `unsigned char` C datatype values into memory.

6.6.3 Data movement pseudo-instructions

The **copy register instruction** (`mv`) is a pseudo-instruction that copies the value from one register into another one. The assembly syntax for this instruction is

```
mv rd, rs
```

where `rd` indicates the target register and `rs` indicates the source register.

The **load immediate instruction** (`li`) is a pseudo-instruction that loads an immediate value into a register. As discussed in Section 6.5.2, depending on the immediate value, the assembler may convert this pseudo-instruction into a single or multiple machine instructions. The assembly syntax for this instruction is

```
li rd, imm
```

where `rd` indicates the target register and `imm` the immediate value to be loaded into the target register.

The **load address instruction** (`la`) is a pseudo-instruction that loads a 32-bit address, indicated by a label, into a register. The assembly syntax for this instruction is

```
la rd, symbol
```

where `rd` indicates the target register and `symbol` the name of the label.

The **load global instructions** are a set of pseudo-instructions to facilitate the load of values from memory positions that are identified by labels. The assembly syntax for these instructions is

```
l{w|h|hu|b|bu} rd, symbol
```

where `rd` indicates the target register and `symbol` the name of the label. As an example, instruction

```
lh a0, var_x
```

loads a `halfword` datatype value from the memory positions starting at the address associated with label "`var_x`". The loaded value is stored into register `a0`. Since a label represents a 32-bit address and may not be encoded as an immediate value on a load instruction[10], the assembler may generate multiple RV32I machine instructions to perform the load operation. In this case, it first generates instructions to load the

[10] The immediate field (`imm`) of load and store instructions are limited to values that can be represented as 12-bit two's-complement numbers.

address of the label into `rd`[11]. Then, it generates a machine load instruction to load the value from memory into `rd`.

The **store global instructions** are a set of pseudo-instructions to facilitate the store of values into memory positions that are identified by labels. The assembly syntax for these instructions is

```
s{w|h|b} rs, symbol, rt
```

where `rs` indicates the source register, `symbol` indicates the name of the label, and `rt` indicates a temporary register to support the computation of the address. As an example, instruction

```
sw a0, var_x, a5
```

stores the `word` value from register `a0` into the memory positions starting at the address associated with label "`var_x`". Similar to the load global instructions, the label may not be encoded as an immediate value, hence, the assembler may generate multiple RV32I machine instructions so that it can load the label address into a register before executing a machine store instruction. In this case, however, the assembler may not use the `rs` as a temporary register, since it would destroy the contents of the register before storing it on the memory. For this reason, the user is required to explicitly indicate a general-purpose register so that the assembler may use it as a temporary when generating the code for the pseudo-instruction.

6.7 Control-flow instructions

In RISC-V, and most general-purpose processors, **the normal execution flow consists in executing instructions in the same order they are organized on memory**. In other words, once an instruction is executed, the processor fetches the instruction that is located in the next memory position for execution. RISC-V instructions are 4-byte long, hence, after executing an instruction that is located at address `0x8000`, the processor fetches the next instruction from address `0x8004`.

Control-flow instructions are instructions capable of changing the normal execution flow. In this case, the next instruction to be executed depends on the semantics of the control-flow instruction.

> **NOTE:** Different authors may use different terms to refer to control-flow instructions. For example, Waterman and Asanović [4] use the term **control transfer instructions**. Other authors may also use the terms **jump** or **branch** to refer to control-flow instructions.

Control-flow instructions may be classified as conditional or unconditional control-flow instructions. They may also be classified as direct or indirect control-flow instructions. The next sections discuss these properties.

6.7.1 Conditional control-flow instructions

Conditional control-flow instructions change the normal execution flow under certain conditions. In other words, the decision of changing or not the normal execution flow depends on whether a given condition is satisfied. For example, the branch equal instruction, or `beq`, compares the values of two registers and **jumps** to the **target address** (*i.e.*, changes the normal execution flow) if their values are equal.

> **NOTE:** The verb "to jump" is commonly used to indicate that a control-flow instruction changed the normal execution flow.

[11] The assembler uses `rd` as a temporary, since its previous value will be discarded anyway after the load operation is complete.

NOTE: The target address is the address of the next instruction that must be fetched in case the instruction jumps.

In the following code, the `beq` instruction jumps to label `L` (which indicates the target address) if the contents of `a0` is equal to the contents of `a1`. In this case, the next instruction to be executed will be the `sub` instruction.

```
1 beq a0, a1, L
2 add a0, a0, a1
3 L:
4 sub a2, a2, a3
```

If the contents of registers `a0` and `a1` differ from each other, then the `beq` instruction does not jump and the execution flow continues normally, *i.e.*, the next instruction on memory (`add`) is executed.

There are several conditional control-flow instructions in the RV32I . Table 6.10 shows the RV32I conditional control-flow instructions and pseudo-instructions.

Instruction	Description
`beq rs1, rs2, lab`	Jumps to label `lab` if the value in `rs1` is equal to the value in `rs2`.
`bne rs1, rs2, lab`	Jumps to label `lab` if the value in `rs1` is different from the value in `rs2`.
`beqz rs1, lab`	Jumps to label `lab` if the value in `rs1` is equal to zero (pseudo-instruction).
`bnez rs1, lab`	Jumps to label `lab` if the value in `rs1` is not equal to zero (pseudo-instruction).
`blt rs1, rs2, lab`	Jumps to label `lab` if the signed value in `rs1` is smaller than the signed value in `rs2`.
`bltu rs1, rs2, lab`	Jumps to label `lab` if the unsigned value in `rs1` is smaller than the unsigned value in `rs2`.
`bge rs1, rs2, lab`	Jumps to label `lab` if the signed value in `rs1` is greater or equal to the signed value in `rs2`.
`bgeu rs1, rs2, lab`	Jumps to label `lab` if the unsigned value in `rs1` is greater or equal to the unsigned value in `rs2`.

Table 6.10: RV32I conditional control-flow instructions

The **blt rs1, rs2, lab** jumps to label `lab` if the value stored on the register indicated by `rs1` is less than the value stored on the register indicated by `rs2`. In this case, the processor assumes that the values in `rs1` and `rs2` are signed values (represented in two's complement), hence, `0xFFFFFFFF`[12] is considered less than `0x00000000` by this instruction. To compare unsigned values, one must use the **bltu rs1, rs2, lab** instruction. In this case, the processor assumes that the values in `rs1` and `rs2` are unsigned binary values, hence, `0xFFFFFFFF`[13] is considered greater than `0x00000000` by this instruction.

The following assembly code shows examples of RV32I conditional control-flow instructions:

```
1 beq  a0, a2, THEN  # Jumps to label THEN if a0  = a2
2 bne  a1, a3, ELSE  # Jumps to label ELSE if a1 != a3
3 blt  a2, a3, NEXT  # Jumps to label NEXT if a2 < a3 (signed comp.)
4 bge  a4, a1, LOOP  # Jumps to label LOOP if a4 >= a1 (signed comp.)
5 bltu a0, a2, L     # Jumps to label L if a0 < a2 (unsigned comp.)
```

[12] `0xFFFFFFFF` represents `-1` in two's complement
[13] `0xFFFFFFFF` represents `4294967295` in the unsigned binary representation

Conditional control-flow instructions are usually employed to implement if and conditional loop statements. The following code shows an example in which the branch not equal instruction (`bne`) is used to implement a loop that iterates 10 times. In this example, register `a0` is used as the loop counter and is initialized with the value ten (line 2). After each loop iteration, the counter is decremented by the subtract (`sub`) instruction (line 7), and the counter is compared with zero by the `bne` instruction[14] (line 8). If the counter is not zero, the control-flow is redirected to the LOOP label and the loop is executed again, otherwise, the execution flow proceeds normally, and the `add` instruction (placed after the loop) is executed (line 9).

```
1 # Initializes the counter
2 mov a0, 10
3 LOOP:
4 # Do something
5 # ...
6 # Decrements the counter and loop back
7 sub a0, a0, 1
8 bne a0, zero, LOOP
9 add a1, a1, a1      # Instruction after the loop
```

Chapter 7 discusses how control-flow instructions can be used to implement high-level language conditional and loop statements, such as C and C++ `if-then`, and `if-then-else` statements and `while` and `for` loops.

> **NOTE:** In the RISC-V ISA manual [4], Waterman and Asanović use the term conditional branch to refer to conditional control-flow instructions. Notice that the term "branch" suggests that the execution may diverge to two different paths, which is the case for RISC-V conditional control-flow instructions. The authors use the term unconditional jumps to refer to unconditional control-flow instructions.

6.7.2 Direct vs indirect control-flow instructions

A direct control-flow instruction is an instruction that has the target address encoded directly into the instruction itself. For example, instruction `beq a0, a1, L` is a direct control-flow instruction because the target address, *i.e.*, the address defined by label `L`, is encoded into the instruction itself.

An indirect control-flow instruction is a control-flow instruction that has the target address specified indirectly, either through memory or through a register. For example, the `jr rs1` is an indirect control-flow instruction that jumps to an address that is defined indirectly by the contents of the register `rs1`. Consequently, the target address of this indirect control-flow instruction can be easily changed by changing the content of `rs1`.

> **NOTE:** Indirect control-flow instructions are often referred to as indirect jump instructions. As we will see in Section 7.3, these instructions can be used to return from functions.

Encoding direct target addresses in RISC-V control-flow instructions

The RV32I encodes memory addresses as 32-bit unsigned values, hence, the **target addresses are naturally 32-bit addresses**. RV32I instructions are all encoded using only 32-bits, hence, it would not be possible to encode a 32-bit target address plus

[14] The pseudo-instruction `bnez a0, LOOP` could also be used in this example, since it is equivalent to the `bne a0, zero, LOOP` instruction.

opcode and other parameters (such as registers) inside a single 32-bit long instruction. To overcome this limitation, direct target addresses are encoded as an offset that is added to the program counter[15] (PC) when the instruction is executed. In RV32I, the target address of conditional control-flow instructions is encoded as 12-bit signed offsets. Using a 12-bit offset to encode the target prevents the instruction from jumping to addresses that are too far away from the address of the instruction itself. In these cases, the programmer loads the target address into a register and, then, use an indirect jump instruction to jump to the target address.

6.7.3 Unconditional control-flow instructions

Unconditional control-flow instructions are control-flow instructions that always change the execution flow to a given target. For example, the "`j L`" instruction is a control-flow instruction that always jumps to label `L`. In the following code, the `j` instruction (line 2) jumps to label , hence, after executing the `j` instruction, the processor executes the `sub` instruction (line 7).

```
1 div a0, a1, a2
2 j   FOO
3 add a0, a0, a1
4 mul a0, a1, a2
5
6 FOO:
7 sub a0, a1, a2
8 # ...
```

> **NOTE:** In the RISC-V ISA manual [4], Waterman and Asanović use the term "unconditional jumps" to refer to unconditional control-flow instructions.

There are several unconditional control-flow instructions in the RV32I . Table 6.11 shows the RV32I unconditional control-flow instructions and pseudo-instructions.

Instruction	Description
`j lab`	Jumps to address indicated by symbol `sym` (pseudo-instruction).
`jr rs1`	Jumps to the address stored on register `rs1` (pseudo-instruction).
`jal lab`	Stores the return address (PC+4) on the return register (`ra`), then jumps to label `lab` (pseudo-instruction).
`jal rd, lab`	Stores the return address (PC+4) on register `rd`, then jumps to label `lab`.
`jarl rd, rs1, imm`	Stores the return address (PC+4) on register `rd`, then jumps to the address calculated by adding the immediate value `imm` to the value on register `rs1`.
`ret`	Jumps to the address stored on the return register (`ra`) (pseudo-instruction).

Table 6.11: RV32I unconditional control-flow instructions

Instruction "`j lab`" jumps to label `lab` while instruction "`jr rs1`" jumps to the address stored on register `rs1`.

Jump and Link

The jump and link instruction (`jal rd, lab`) stores the address of the subsequent instruction (*i.e.*, PC+4) in register `rd` and then jumps to label `lab`. This process,

[15] The program counter contains the address of the currently executing instruction. When executing a control-flow instruction it contains the address of the said instruction.

known as jump and link, is particularly useful when invoking routines. The following code, annotated with addresses, illustrates this process: First, the `jal` instruction (located at memory address `0x8000`) is used to invoke (jump to) routine FOO (line 1). When executed, this instruction will store the address of the subsequent instruction (PC+4 = `0x8004`) into register `ra` and jump to FOO. Then, once FOO is executed, the routine returns by performing an indirect jump (`jr`) to the contents of register `ra` (line 8). At this point, since `ra` contains the value `0x8004`, the execution flows back to the instruction `sub` (line 2). Next, another jump and link instruction is executed to invoke routine FOO. Again, the jump and link instruction will store the address of the subsequent instruction (PC+4 = `0x800C`) into register `ra` and jump to FOO. Notice, however, that at this time, the subsequent instruction is the `mul` instruction, and the return address is `0x800C`. Finally, once FOO is executed, the routine returns by performing an indirect jump (`jr`) to the contents of register `ra` (line 8), which in this case contains the value `0x800C` and will cause the execution to flow back to instruction `mul`.

```
1 0x8000: jal ra, FOO      # Invoke routine FOO
2 0x8004: sub a0, a1, a2
3 0x8008: jal ra, FOO      # Invoke routine FOO again
4 0x800C: mul a0, a1, a2
5
6 FOO:                     # Routine foo
7 0x8080: add a0, a0, a1   # Perform some computation
8 0x8084: jr  ra           # Returns from routine
```

The "`jarl rd, rs1, imm`" instruction also stores the address of the subsequent instruction (*i.e.*, PC+4) in register `rd`, however, in this case, the target address is defined by adding the contents of `rs1` to the immediate `imm`. Hence, it is an indirect jump.

NOTE: The jump register (`jr`) and return (`ret`) instructions are pseudo-instructions that are converted into `jarl` instructions by the assembler. The `jr rs1` is converted into `jarl zero, rs1, 0` and `ret` is converted into `jarl zero, ra, 0`. Notice that, in RV32I, any value stored in register `zero` is discarded.

NOTE: The "`j lab`" instruction is a pseudo-instruction that is converted by the assembler into `jal zero, lab`. Notice that, in RV32I, any value stored in register `zero` is discarded.

6.7.4 System Calls

User programs usually need to perform input and output operations. For example, after performing some computation, the program may need to print the result on the screen, or write it to a file.

Input and output operations are usually performed by input and output devices, such as keyboards, pointing devices, printers, display, network devices, hard drives, *etc.*. These devices are managed by the operating system, hence, input and output operations are also performed by the operating system. To increase portability and facilitate software development, operating systems usually implement a set of abstractions (such and folders, files, *etc.*) and offer a set of service routines to allow user programs to perform input and output operations. In this context, user programs perform input and output operations by invoking the operating system service routines.

A system call, or syscall, is a special call operation used to invoke operating system service routines. In RISC-V, this operation is performed by a special instruction called `ecall`.

As an example, let us assume the user wants to invoke the Linux operating system service routine "write" (also known as write syscall) to display some information on the screen. The write syscall takes three parameters: the file descriptor, the address of a buffer that contains the information that must be written, and the number of bytes that must be written. The file descriptor is an integer value that identifies a file[16] or a device. In this case, it indicates the file or device where the information must be written to. In many Linux distributions, the file descriptor '1' (one) is used to represent the standard output, or `stdout`, which is usually the terminal screen. The following code shows an example in which the contents of the `msg` buffer is written to the file descriptor '1', hence, the screen. First, the code sets the system call parameters (lines 6 to 8), then it sets register `a7` with a number that indicates the service routine that must be invoked, which, in this case, is the write syscall (line 9). Finally, it invokes the operating system by executing the `ecall` instruction.

```
1  .data
2  msg: .asciz "Assembly rocks" # Allocates a string on memory
3
4  .text
5  start:
6   li a0, 1     # a0: File descriptor = 1 (stdout)
7   la a1, msg   # a1: Message buffer address
8   li a2, 14    # a2: Message buffer size (14 bytes)
9   li a7, 64    # Syscall code (write = 64)
10  ecall        # Invoke the syscall
```

NOTE: Each operating system may have a different set of service routines. The focus of this book is not to discuss the set of service routines offered by an specific operating system; nonetheless, to illustrate the concepts, whenever necessary, it will use service routines available on the Linux operating system.

6.8 Conditional set instructions

Similar to a conditional control-flow instruction, a conditional set instruction compares two values, however, instead of jumping to a label, it sets the target register with values one or zero, indicating whether or not the condition is true. Table 6.12 shows the RV32I conditional set instructions and pseudo-instructions.

The set less than instruction (`slt`) places the value 1 in register `rd` if the signed value in register `rs1` is less than the signed value in register `rs2`, else 0 is written to register `rd`. Notice that, in this case, both values are treated as signed numbers. The `sltu` instruction is similar, however, it compares the values as unsigned numbers.

The set less than immediate instruction (`slti`) places the value 1 in register `rd` if the signed value in register `rs1` is less than the sign-extended immediate, else 0 is written to register `rd`. Notice that, in this case, both values are treated as signed numbers. The `sltiu` instruction is similar, however, it compares the values as unsigned numbers.

6.9 Detecting overflow

The RISC-V does not provide special hardware support to detect overflow when performing arithmetic operations. However, regular conditional branch and conditional set instructions may be used to find out whether or not an overflow occurred. The following code shows how to detect whether or not an overflow occurred when adding

[16] The system call `open` may be used to open files, in this case, it returns the file descriptor of the opened file.

Instruction	Description
`slt rd, rs1, rs2`	Sets `rd` with 1 if the signed value in `rs1` is less than the signed value in `rs2`, otherwise, sets it with 0.
`slti rd, rs1, imm`	Sets `rd` with 1 if the signed value in `rs1` is less than the sign-extended immediate value `imm`, otherwise, sets it with 0.
`sltu rd, rs1, rs2`	Sets `rd` with 1 if the unsigned value in `rs1` is less than the unsigned value in `rs2`, otherwise, sets it with 0.
`sltui rd, rs1, imm`	Sets `rd` with 1 if the unsigned value in `rs1` is less than the unsigned immediate value `imm`, otherwise, sets it with 0.
`seqz rd, rs1`	Sets `rd` with 1 if the value in `rs1` is equal to zero, otherwise, sets it with 0 (pseudo-instruction).
`snez rd, rs1`	Sets `rd` with 1 if the value in `rs1` is not equal to zero, otherwise, sets it with 0 (pseudo-instruction).
`sltz rd, rs1`	Sets `rd` with 1 if the signed value in `rs1` is less than zero, otherwise, sets it with 0 (pseudo-instruction).
`sgtz rd, rs1`	Sets `rd` with 1 if the signed value in `rs1` is greater than zero, otherwise, sets it with 0 (pseudo-instruction).

Table 6.12: RV32I conditional set instructions

two unsigned integers. In this case, the `bltu` instruction jumps to label `handle_ov` in case there is an overflow.

```
1 add  a0, a1, a2        # Add two values
2 bltu a0, a1, handle_ov # Jumps to handle_ov in case
3                        # an overflow happened
4 ...
5 handle_ov:             # Code to handle the overflow
6 ...
```

Alternatively, instead of jumping to a label to handle the overflow, the following code can be used to set a register (`t1` in this case) to indicate whether or not occurred an overflow. Notice that the `sltu` instruction sets register `t1` with one in case the contents of `a0` (which contains `a1`+`a2`) is less than the contents of `a1`.

```
1 add  a0, a1, a2     # Add two values
2 sltu t1, a0, a1     # t1 <= 1 if (a1+a2) < a1 (Overflow)
3                     # otherwise, t2 <= 0 (No overflow)
```

The following code shows how to detect overflow when adding two signed numbers.

```
1 add  a0, a1, a2        # Add two values
2 slti t1, a2, 0         # t1 = (a2 < 0)
3 slt  t2, a0, a1        # t2 = (a1+a2 < a1)
4 bne  t1, t2, handle_ov # overflow if  (a2<0) && (a1+a2>=a1)
5                        #       or if (a2>=0) && (a1+a2<a1)
6 ...
7 handle_ov:             # Code to handle the overflow
8 ...
```

6.10 Arithmetic on multi-word variables

The RV32I has natural hardware support to perform arithmetic operations on 32-bit values. The `add` instruction, for example, can be used to add two (signed or unsigned)

32-bit numbers. However, in some situations, it may be necessary to add multi-word (more than 32-bit) values. In these cases, the user may add the 32-bit parts and account for the carry-outs and carry-ins across these parts.

The following examples show how two 64-bit values can be added together. The first value is stored at register pair `a1:a0`[17] while the second value is stored in register pair `a3:a2`. Also, the results are being placed in the register pair `a5:a4`. The first step is to add the least significant words (line 1). Then to account for the carry-out, the code checks whether the result of the unsigned addition is less than one of the operands (`a2` in this case). If so, it means that there was a carry-out, and the register `t1` is set with 1. Otherwise, there was no carry-out, and the register `t1` is set with 0 (line 2). Next, the most significant words are added together (line 4), and, finally, the carry-out is also added to the most significant words (line 5).

```
1 add  a4, a0, a2 # add the least significant word
2 sltu t1, a4, a2 # t1 <= 1 if (a0+a2) < a2 (Overflow = carry out)
3                 # otherwise, a2 <= 0
4 add  a5, a1, a3 # Add the most significant word
5 add  a5, t1, a5 # Add the carry out
```

[17] This notation is used to indicate that two registers are being used to store a 64-bit value. The least significant value is being stored by register `a0` while the most significant value is being stored by register `a1`.

Chapter 7

Controlling the execution flow

This chapter discusses how to implement in assembly high-level language conditional and loop statements, such as C and C++ `if-then`, and `if-then-else` statements and `while` and `for` loops. It also discusses how to invoke and return from functions.

7.1 Conditional statements

7.1.1 `if-then` statements

The following code shows an `if-then` statement written in "C". In this code, the value of `x` is assigned to variable `y` if the value of `x` is greater or equal to ten. The `if-then` statement contains two main parts: the "**condition**" (line 1) and the **"then block"** (lines 2-4). The condition consists of a boolean expression that must be evaluated first and the "then block" contains a set of statements that must be executed in case the boolean expression is true.

```
1 if (x >= 10)
2 {
3  y = x;
4 }
```

Assuming `x` is a signed integer (`int`) mapped to register `a3`[1] and `y` is mapped to register `a4`, the following code shows how the previous "C" code can be implemented in assembly language. First, the code loads the constant ten into temporary register `t1` (line 1). Then, if the contents of register `a3` (variable `x`) are less than ten[2], it jumps to label `skip`, otherwise, it executes the next instruction (line 3), which corresponds to the "then block".

```
1   li  t1, 10
2   blt a3, t1, skip # jumps to skip if x < 10
3   mv  a4, a3       # y = x
4 skip:
```

In case there are multiple statements in the "then block", they can be placed between the branch less than instruction (line 2) and the `skip` label (line 4).

7.1.2 Comparing signed vs unsigned variables

The previous code used the instruction `blt` to verify if the contents of variable `x` is less than 10. As discussed in Section 6.7.1, this is correct because variable `x` is a

[1]When converting "C" programs to assembly language, the programmer, or the compiler, usually map variables to machine registers or to memory positions.

[2]Register `t1` contains the value ten at this point

signed variable. In case variable `x` was an unsigned integer (`unsigned` "C" type), the programmer (or the compiler) must have used instruction `bltu` to perform the comparison.

> **NOTE:** It is important to notice that the processor does not automatically infer if the contents of a register (or memory) is a signed or unsigned value. The programmer (or the compiler) must inform the processor by selecting the proper instruction to perform the comparison - in the previous example, `blt` for signed variables and `bltu` for unsigned variables.

7.1.3 `if-then-else` statements

The following code shows an `if-then-else` statement written in "C". In this code, if the value of `x` is greater or equal to ten, then the value of variable `y` is incremented by one, else, the the value of `x` is assigned to variable `y`. The `if-then-else` statement contains three main parts: the "**condition**" (line 1), the **"then block"** (lines 2-4), and the **"else block"** (lines 6-8). The condition consists of a boolean expression that must be evaluated first. The "then block" contains a set of statements that must be executed in case the boolean expression evaluates to true, and the "else block" contains a set of statements that must be executed in case the boolean expression evaluates to false.

```
1 if (x >= 10)
2 {
3  y = y + 1;
4 }
5 else
6 {
7  y = x;
8 }
```

Assuming `x` is an unsigned integer (`unsigned` "C" type) mapped to register `a1` and `y` is mapped to register `a2`, the following code shows how the previous "C" code can be implemented in assembly language. First, the code loads the constant ten into temporary register `t3` (line 1). Then, if the contents of register `a1` (variable `x`) are less than ten[3], it jumps to label `else` to execute the "else block", otherwise, it executes the next instruction (line 3), which corresponds to the first instruction of the "then block".

```
1   li    t3, 10
2   bltu  a1, t3, else # jumps to else if x < 10
3   addi  a2, a2, 1    # y = y + 1
4   j     cont         # skip the else block
5 else:
6   mv   a2, a1        # y = x
7 cont:
```

In case there are multiple statements in the "then block", they can be placed between the instruction `bltu` (line 2) and the `j cont` instruction (line 4). In case there are multiple statements in the "else block", they can be placed between labels `else` (line 5) and `cont` (line 7).

[3]Register `t3` contains the value ten at this point

7.1.4 Handling non-trivial boolean expressions

In some cases, the boolean expression being evaluated by the conditional statement may contain multiple operations. The following code shows an example in which the boolean expression contains multiple operations. In this case, the "then block" can only be executed if the contents of variable `x` is greater or equal to 10 **and** if the contents of variable `y` is less than 20.

```
1 if ((x>=10) && (y<20))
2 {
3   x = y;
4 }
```

Assuming `x` and `y` are a signed integer variables (`int` "C" type) mapped to registers `a1` and `a2`, respectively, the following code shows how the previous "C" code can be implemented in assembly language. First, the code loads the constant ten into temporary register `t1` (line 1). Then, if the contents of register `a1` (variable `x`) are less than ten, it jumps to label `skip` to skip the execution of the "then block". Notice that, because of the **and** operator (represented by `&&` in "C"), if the first part of the boolean expression is false, then the whole expression is false, hence, there is no need to check the second part[4]. If the first part of the boolean expression (lines 1 and 2) has been evaluated to true, then the code must check the second part, which is verified by instructions in lines 3 and 4. In this case, if the contents of variable `y` is greater or equal to 20, then the code (line 4) skips the "then block" by jumping to the `skip` label. Otherwise, it executes the next instruction (line 5), which corresponds to the first instruction of the "then block".

```
1  li   t1, 10
2  blt  a1, t1, skip # jumps to skip if x < 10
3  li   t1, 20
4  bge  a2, t1, skip # jumps to skip if y >= 20
5  mv   a1, a2       # x = y
6 skip:
```

The following code shows an example in which the boolean expression contains an **or** (`||`) operation.

```
1 if ((x>=10) || (y<20))
2 {
3   x = y;
4 }
```

Assuming `x` and `y` are signed integer variables (`int` "C" type) mapped to registers `a1` and `a2`, respectively, the following code shows how the previous "C" code can be implemented in assembly language. First, the code loads the constant ten into temporary register `t1` (line 1). Then, if the contents of register `a1` (variable `x`) are greater or equal to ten, it jumps to label `then` to execute the "then block". Notice that, because of the **or** operator (represented by `||` in "C"), if the first part of the boolean expression evaluates to true, then the whole expression is true, hence, there is no need to check the second part[5]. If the first part of the boolean expression (lines 1 and 2) evaluates to false, then the code must check the second part, which is verified by instructions in lines 3 and 4. In this case, if the contents of variable `y` is greater or equal to 20, then the code (line 4) skips the "then block" by jumping to the `skip`

[4]The semantics of the "C" programming language implies that the remaining of this boolean expression must not be evaluated if the first part is false.

[5]The semantics of the "C" programming language implies that the remaining of this boolean expression must not be evaluated if the first part is true.

label. Otherwise, it executes the next instruction (line 5), which corresponds to the first instruction of the "then block".

```
1  li   t1, 10
2  bge  a1, t1, then # jumps to then if x >= 10
3  li   t1, 20
4  bge  a2, t1, skip # jumps to skip if y >= 20
5 then:
6  mv   a1, a2       # x = y
7 skip:
```

7.1.5 Nested if statements

Nested `if-then` and `if-then-else` are if statements in which the "then block" or the "else block" contain other if statements. The following code shows an example of a nested `if-then` statement. Notice that there are two `if-then` statements: an outer one (lines 1-8) and an inner one (lines 4-7). The "then block" of the outer `if-then` statement contains two statements: a variable assignment (line 3) and the inner `if-then` statement (lines 4-7).

```
1 if (x == 10)
2 {
3   x = 5;
4   if (y == 20)
5   {
6     x = 0;
7   }
8 }
```

Translating the previous code to assembly code can be easily done by starting with the outer `if-then` statement. Assuming `x` and `y` are variables mapped to registers `a1` and `a2`, respectively, the following code shows the skeleton for the outer `if-then` statement.

```
1  li   t1, 10
2  bne  a1, t1, skip # jumps to skip if x != 10
3                    # <= Insert the code for the then block here
4 skip:
```

Once the skeleton code is generated, the next step is to generate the code for the "then block". The following code shows the final code.

```
1  li   t1, 10
2  bne  a1, t1, skip       # jumps to skip if x != 10
3  li   a1, 5              # x = 5
4  li   t1, 20
5  bne  a2, 20, skip_inner # jumps to skip_inner if y != 20
6  li   a1, 0              # x = 0
7 skip_inner:
8 skip:
```

In the previous example, we used two different labels, one to skip the execution of the "then block" of the outer if statement (`skip`) and another to skip the execution of the "then block" of the inner if statement (`skip_inner`). In this case, since both labels represent the same address (notice that there are no instructions or data between both

labels), we could simplify the code by using a single skip label. Notice, however, that this simplification does not affect the code generated by the assembler, since labels are basically addresses annotations.

7.2 Repetition statements

Repetition statements, or loops, are used to repeat a group of statements. In this section, we discuss how to implement the most common repetition statements in assembly.

7.2.1 `while` loop

The following code shows a `while` loop in "C". The `while` loop contains two main parts: the **"loop condition"** (line 2) and the **"loop body"** (lines 3-6). The condition consists of a boolean expression that must be evaluated before each iteration of the `while` loop, *i.e.*, before each execution of the "loop body". If the "loop condition" evaluates to true, then the "loop body" must be executed. In this case, after completing the execution of the "loop body", the execution jumps back to the beginning of the loop and the "loop condition" is evaluated again. If the "loop condition" evaluates to false, then the loop execution is complete and the execution continues after the loop body.

```
1 int i=0;
2 while (i < 20)
3 {
4   y = y+3;
5   i = i+1;
6 }
```

Assuming that variables `i` and `y` are mapped to registers `a1` and `a2`, respectively, the following code shows how the previous "C" code can be implemented in assembly language. First, it contains the code that comes before the while loop - in this case an instruction that loads the constant zero into register `a1` (line 1). Then, there is a label that defines the beginning of the loop (line 2) and the code that checks the "loop condition" (lines 3 and 4). Notice that instruction `bge` checks whether variable `i` (contents of register `a1`) is greater or equal to 20. If so, it jumps to label `skip` to leave the loop. Otherwise, the execution continues on the next instruction, which is the first instruction of the "loop body". The "loop body" in this example is composed by two instructions (lines 5 and 6). The first one implements the statement `y=y+3` and the second one implements the statement `i=i+1`. After concluding the execution of the "loop body", the code jumps back to the beginning of the loop (line 7) so that the loop can be executed again, starting by verifying again the "loop condition".

```
1   li   a1, 0        # i=0
2 while:
3   li   t1, 20       # if i>=20
4   bge  a1, t1, skip # jump to skip to leave the loop
5   addi a2, a2, 3    # y = y+3
6   addi a1, a1, 1    # i = i+1
7   j    while        # loop back
8 skip:
```

7.2.2 `do-while` loop

The following code shows a `do-while` loop written in "C". Similar to the `while` loop, the `do-while` loop contains two main parts: the **"loop condition"** (line 6) and

the **"loop body"** (lines 3-5). The condition also consists of a boolean expression, however, int the `do-while` statement, the condition must be evaluated after each iteration of the `do-while` loop, *i.e.*, after each execution of the "loop body". If the "loop condition" evaluates to true, then the "loop body" must be executed again. In this case, after completing the execution of the "loop body", the "loop condition" is evaluated again. If the "loop condition" evaluates as false, then the loop execution is complete and the execution continues after the loop.

```
1 int i=0;
2 do
3 {
4   y = y+2;
5   i = i+1;
6 } while (i < 10);
```

Assuming that variables `i` and `y` are mapped to registers `a1` and `a2`, respectively, the following code shows how the previous "C" code can be implemented in assembly language. First, it contains the code that comes before the while loop - in this case an instruction that loads the constant zero into register `a1` (line 1). Then, there is a label that marks the beginning of the loop (line 2) and the "loop body", which is composed of two instructions (lines 5 and 6). The first one implements the statement `y=y+2` and the second one implements the statement `i=i+1`. After the "loop body", there is the code that checks the "loop condition" (lines 5 and 6). Notice that instruction `blt` checks whether variable `i` (contents of register `a1`) is less than 10. If so, it jumps back to label `dowhile` to repeat the loop execution. Otherwise, the execution continues on the next instruction, leaving the loop.

```
1   li   a1, 0           # i=0
2 dowhile:
3   addi a2, a2, 2       # y = y+2
4   addi a1, a1, 1       # i = i+1
5   li   t1, 10
6   blt  a1, t1, dowhile # jumps back to dowhile if i < 10
```

7.2.3 `for` loop

The following code shows a `for` loop written in "C". The `for` loop contains four main parts: the **"initialization code"**, the **"loop condition"**, the **"update code"** and the **"loop body"** (lines 2-4). The "initialization code" (`i=0`) must be executed exactly once before the execution of the "loop body". Similar to the `while` loop, the "loop condition" (`i<10`) consists of a boolean expression that must be evaluated before each iteration of the `for` loop, *i.e.*, before each execution of the "loop body" (lines 2-4). If the "loop condition" evaluates to true, then the "loop body" must be executed. In this case, after completing the execution of the "loop body", the "update code" (`i=i+1`) is executed and the execution jumps back to the beginning of the loop so that the "loop condition" can be evaluated again. If the "loop condition" evaluates to false, then the loop execution is complete and the execution continues after the loop body.

```
1 for (i=0; i<10; i=i+1)
2 {
3   y = y+2;
4 }
```

Assuming that variables `i` and `y` are mapped to registers `a1` and `a2`, respectively, the following code shows how the previous "C" code can be implemented in assembly

language. First, it contains the "initialization code", *i.e.*, `i=0` (line 1). Then, there is a label that defines the beginning of the loop (line 2) and the code that checks the "loop condition" (lines 3 and 4). Notice that instruction `bge` checks whether variable `i` (contents of register `a1`) is greater or equal to 10. If so, it jumps to label `skip` to leave the loop. Otherwise, the execution continues on the next instruction, which is the first instruction of the "loop body". The "loop body" in this example is composed of only one instruction (line 5), which implements the statement `y=y+2`. The "update code", *i.e.*, `i=i+1` (line 6), is placed right after the "loop body". Finally, after the "update code", the code jumps back to the beginning of the loop (line 7) so that the loop can be executed again, starting by verifying again the "loop condition".

```
1   li   a1, 0        # i=0
2 for:
3   li   t1, 10       # if i >= 10 then jumps
4   bge  a1, t1, skip # to skip to leave the loop
5   addi a2, a2, 2    # y = y+2
6   addi a1, a1, 1    # i = i+1
7   j    for
8 skip:
```

7.2.4 Hoisting loop-invariant code

Loop-invariant codes are those that always produce the same values and does not need to be re-executed every time a "loop body" is repeated. In the following example, the instruction `li t1,10` is a loop-invariant code and should not be executed every time the "loop body" is repeated.

```
1   li   a1, 0        # i=0
2 for:
3   li   t1, 10       # if i >= 10 then jumps
4   bge  a1, t1, skip # to skip to leave the loop
5   addi a2, a2, 2    # y = y+2
6   addi a1, a1, 1    # i = i+1
7   j    for
8 skip:
```

In these cases, the instruction can be hoisted (moved) before the loop to improve the code performance. This is an optimization called "loop-invariant code motion" (LICM) commonly applied by compilers. The following code shows the code after applying LICM to the previous code. Notice that, in this case, each loop repetition executes only four instructions.

```
1   li   a1, 0        # i=0
2   li   t1, 10       # t1=10
3 for:
4   bge  a1, t1, skip # if i >= 10 then jumps to skip to leave the loop
5   addi a2, a2, 2    # y = y+2
6   addi a1, a1, 1    # i = i+1
7   j    for
8 skip:
```

7.3 Invoking and returning from routines

Routines are defined in assembly language by a label and a fragment of code. The label defines the routine entry point, *i.e.*, the point to which the execution must flow

when the routine is invoked. The label usually defines the routine name[6] and the fragment of code contains the instructions that implement the routine. The following code shows an example of a routine called `update_x`. This routine stores in variable `x` the value of from register `a0` and then returns.

```
1 # The update_x routine
2 update_x:
3   la   t1, x
4   sw   a0, (t1)
5   ret
```

Invoking a routine is as simple as jumping to the label that defines its entry point. However, **before invoking (jumping to) the routine, it is important to save the return address so that the routine can return to the calling site after its execution**.

As discussed in Section 6.7.3, RV32I contains a special jump instruction to facilitate saving the return address when invoking a routine. This instruction, called **jump and link**, or `jal lab`, stores the return address[7] (PC+4) on the return address register (`ra`) and then jumps to the label `lab`.

The following fragment of code shows how to invoke the `update_x` routine to update the value of variable `x` with value 42. First, it loads value 42 into register `a0`, then it invokes the `update_x` routine using the jump and link (`jal`) instruction.

```
1 .data
2 x: .skip 4
3
4   li  a0, 42    # loads 42 into a0
5   jal update_x  # invoke the update_x routine
```

When the `update_x` routine finishes executing, it needs to return to the calling site. This operation can be performed by jumping to the address stored in register `ra`, which was set by the `jal` instruction when the routine was invoked. The pseudo-instruction `ret` performs this operation.

> **NOTE:** As discussed above, the return address is automatically stored at register `ra` by the `jal` instruction. This operation destroys the previous contents of register `ra`, hence, before invoking a routine, it may be necessary to save the contents of this register so that it may be recovered later. This is especially important when invoking a routine B from within another routine A since the contents of register `ra` must be preserved so that routine A can return after its completion. Chapter 8 discusses how to save and recover the contents of `ra` when invoking routines.

7.3.1 Returning values from functions

A **function** is a routine that can accept arguments and returns one or more values. A **procedure**, on the other hand, is a routine that can accept arguments but does not return any values.

Returning a value from a function is a matter of convention and is usually defined by the Application Binary Interface, or ABI. The RISC-V ABI defines that functions must return values by storing them on register `a0`. Section 8.3 further discusses the Application Binary Interface and its importance for software composition.

[6] When translating "C" code into assembly code, the routine entry-point labels are exactly the same as the routine name.

[7] The return address is the address of the next instruction on memory, *i.e.*, the address of the current instruction (PC) plus four.

7.4 Examples

7.4.1 Searching for the maximum value on an array

The following "C" code shows a global array variable named `numbers` and a function that returns the largest value from the array.

```
/* Global array */
int numbers[10];

/* Returns the largest value from array numbers. */
int get_largest_number()
{
    int largest = numbers[0];
    for (int i=1; i<10; i++) {
        if (numbers[i] > largest)
            largest = numbers[i];
    }
    return largest;
}
```

```
.data
# Allocate the numbers array (10 integers = 40 bytes)
numbers: .skip 40

.text
get_largest_number:
  la a5, numbers     # a5 = &numbers
  lw a0, (a5)        # a0 (largest) = numbers[0]
  li a1, 1           # a1 (i) = 1
  li t4, 10
  for:
   bge  a1, t4, end: # if i >= 10, then end loop
   slli t1, a1, 2    # t1 = i * 4
   add  t2, a5, t1   # t2 = &numbers + i*4
   lw   t3, (t2)     # t3 = numbers[i]
   blt  t3, a0       # if numbers[i] < largest, then skip
   mv   a0, t3       # Update largest
   skip:
   addi a1, a1, 1  # i = i+1
   j    for
  end:
  ret                # Return
```

Alternative solutions:

```
get_largest_number:
        lui  a5,%hi(numbers)
        lw   a0,%lo(numbers)(a5)
        addi a5,a5,%lo(numbers)
        addi a3,a5,36
.L3:
        lw   a4,4(a5)
        addi a5,a5,4
        bge  a0,a4,.L2
        mv   a0,a4
.L2:
```

```
12          bne  a5,a3,.L3
13          ret
```

```
1  get_largest_number:
2    la   a5, numbers      # a5: pointer to current element
3    lw   a0, (a5)         # Load first element (number[0])
4    addi a5, a5, 4        # Advance pointer to next element
5    addi a6, a5, 40       # a6 <= address after last element
6  do_while:
7    lw   a4, (a5)         # Load current element
8    bge  a0, a4, skip
9    mv   a0, a4           # Update largest
10 skip:
11   addi a5, a5, 4        # Advance pointer to next element
12   bne  a5, a3, do_while # do while a5 != a3
13   ret
```

Chapter 8

Implementing routines

8.1 The program memory layout

A **stored-program** computer architecture is a computer architecture that stores both data and code on memory. A **Von Neumann architecture** is a computer architecture that stores both the data and the code at the same address space. Most modern computer architectures are Von Neumann architectures.

Figure 8.1 shows a common way of organizing programs on the memory of Von Neumann architectures. The **code space is a memory space that stores the program code** and is usually placed first, in the lowest addresses. The **static data space is a memory space that stores the program static data** (*e.g.*, global variables) and is placed after the code. The **heap space is a memory space managed by the memory allocation library**[1] and is allocated after the static data. The heap starts small and, whenever the memory allocation library needs more space, it invokes the operating system to grow the heap area, which does it by increasing the "program break" address[2]. Finally, the **stack space is a memory space that stores the program stack** and is usually placed at the end (high addresses) of the memory.

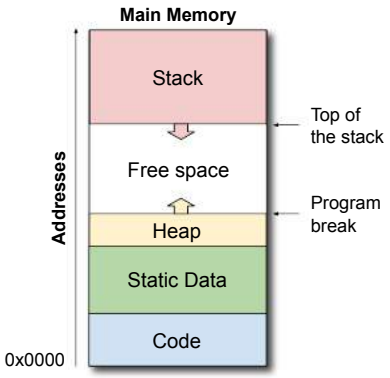


Figure 8.1: Common program memory layout.

8.2 The program stack

An **active routine is a routine that was invoked but did not return yet**. Notice that there may be multiple active routines at a given point of the execution.

[1]The memory allocation library allows the program to dynamically allocate (and free) data on the heap. Data allocated using the `malloc` routine, for example, is allocated into the heap. The memory allocation library is responsible for keeping track of which addresses within the heap are free or allocated.

[2]The program break defines the end of the heap.

For example, in the following code, routine `fun` is invoked by routine `bar`, which is invoked by routine `main`. Initially, the `main` routine is active. Then, it invokes the `bar` routine, which also becomes active. Finally, the `fun` routine is invoked and it also becomes active. At this point, there are three active routines in the system.

```
1  int a = 10;
2
3  int main()
4  {
5      return bar() + 2;
6  }
7  int bar()
8  {
9      return fun() + 4;
10 }
11 int fun()
12 {
13     return a;
14 }
```

The set of active routines increases whenever a routine is invoked and decreases whenever a routine returns. Routines are activated and deactivated in a last-in-first-out fashion, *i.e.*, the last one to be activated must be the first one to be deactivated. Consequently, **the most natural data structure to keep track of active routines is a stack**.

Routines usually need memory space to store important information, such as local variables, parameters, and the return address. Hence, whenever a routine is invoked (and becomes active), the system needs to allocate memory space to store information related to the routine. Once it returns (is deactivated), all the information associated with the routine invocation is not needed anymore and this memory space must be freed.

The **program stack is a stack data structure that stores information about active routines**, such as local variables, parameters, and the return address. The program stack is stored in the main memory and, whenever a routine is invoked, the information about the routine is pushed on top of the stack, which causes it to grow. Also, whenever a routine returns, the information about the routine is discarded by dropping the contents at the top of the stack, which causes it to shrink.

The program stack is allocated at the stack space, which is usually placed at the end (high addresses) of the memory. As a consequence, the program stack must grow towards low addresses.

The stack pointer is a pointer to the top of the stack, *i.e.*, it stores the address of the top of the stack. Growing or shrinking the stack is performed by adjusting the stack pointer.

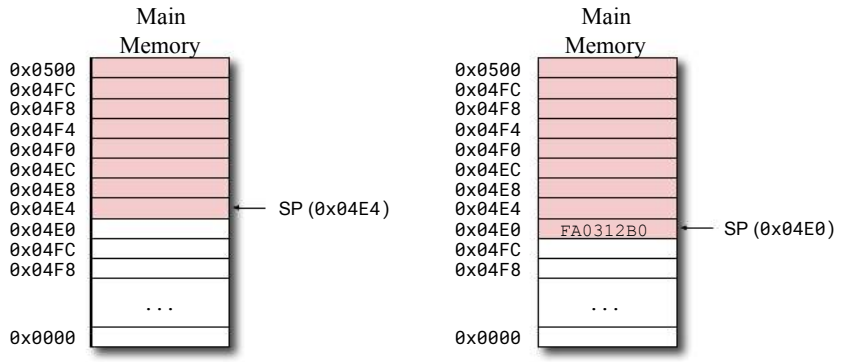
In RISC-V, the **stack pointer is stored by register** `sp`. Also, in RISC-V, the stack grows towards low addresses, hence, growing (or allocating space on) the stack can be performed by decreasing the value of register `sp` (the stack pointer). The following code shows how to push the contents of register `a0` into the stack. First, the stack pointer is decreased to allocate space (4 bytes), then, the contents of register `a0` (4 bytes) are stored on the top of the program stack using the `sw` instruction.

```
1 addi sp, sp, -4  # allocate stack space
2 sw   a0, 0(sp)   # store data into the stack
```

Alternatively, shrinking the stack can be performed by increasing the value of register `sp`. The following code shows how to pop a value from the top of the stack into register `a0`. First, the value on the top of the program stack is loaded into register `a0` (4 bytes) using the `lw` instruction. Then, the stack pointer is increased to deallocate the space (4 bytes).

```
lw   a0, 0(sp)   # retrieve data from stack
addi sp, sp, 4   # deallocate space
```

Figure 8.2 (a) illustrates a program stack that starts at address `0x0500` and grows down to address `0x04E4`. Since the stack pointer points to the top of the stack, the contents of the register `sp` is equal to `0x04E4`.



(a) (b)

Figure 8.2: Example of a program stack starting at address `0x0500` and growing downward to address `0x04E4` (a) before and (b) after pushing value `0xFA0312B0` into the stack.

Figure 8.2 (b) shows how the program stack is modified after executing the following code, *i.e.*, after pushing the contents of register `a0` into the stack. Notice that the value of the stack pointer (`sp`) was decremented by 4 units and the contents of register `a0` (`0xFA0312B0`) was stored at the memory starting at address `0x04E0`.

```
li   a0, 0xFA0312B0
addi sp, sp, -4     # allocate stack space
sw   a0, 0(sp)      # store data into the stack
```

The previous examples discussed how to push and pop a single word (4-byte value) to and from the stack. In many situations, a program may need to push or pop multiple values to or from the stack. For example, the program may need to save a set of register values on the stack. In these cases, the code may be optimized by adjusting (increasing or decreasing) the stack pointer only once. The following code shows how to push four values from registers `a0`, `a1`, `a2`, and `a3` into the program stack. Notice that the stack pointer was adjusted only once and the immediate field of the store word instruction (`sw`) was used to select the proper position to store each one of the values. In this example, the last value pushed into the stack was the value stored in the register `a3`.

```
addi sp, sp, -16  # allocate stack space (4 words)
sw   a0, 12(sp)   # store the first value (SP+12)
sw   a1,  8(sp)   # store the second value (SP+8)
sw   a2,  4(sp)   # store the third value (SP+4)
sw   a3,  0(sp)   # store the fourth value (SP+0)
```

The following code shows how to pop four values from the program stack into registers `a3`, `a2`, `a1`, and `a0`. Notice that the stack pointer was adjusted only once and the immediate field of the load word instruction (`lw`) was used to select the proper position to load each one of the values. In this example, the first value popped from the stack was stored into register `a3`.

```
1 lw   a3,  0(sp)   # retrieve the first value (SP+0)
2 lw   a2,  4(sp)   # retrieve the second value (SP+4)
3 lw   a1,  8(sp)   # retrieve the third value (SP+8)
4 lw   a0, 12(sp)   # retrieve the last value (SP+12)
5 addi sp, sp, 16   # deallocate stack space (4 words)
```

Previous examples showed how to perform push and pop operations on the program stack. Popping data from the stack consists of retrieving the data and then deallocating the stack space, however, if the data is not needed anymore, then, only the deallocation process needs to be performed. As we will discuss in sections 8.6 and 8.4, allocating or deallocating stack space without immediately storing or retrieving data to or from the stack is useful in many cases.

Initializing the stack pointer

The stack pointer register must be initialized to the base of the program stack before executing the program. When running the program without the support of an operating system (for example, in an embedded system) the stack pointer must be initialized by the system initialization code. On the other hand, when running the program on top of an operating system, the execution environment (*e.g.*, the operating system kernel) usually initializes the stack pointer before jumping to the program's entry point.

8.2.1 Types of stacks

Program stacks can be classified as: full descending, full ascending, empty descending, and empty ascending.

A **full descending stack is a stack that: (a) grows towards low addresses, and (b) the stack pointer points to the last element pushed into the stack**. Pushing a value into a full descending stack is performed by decreasing the stack pointer and then storing the value into the memory word pointed by the stack pointer. Popping a value from a full descending stack is performed by first loading the value from the memory word pointed by the stack pointer and then increasing the stack pointer.

An **empty descending stack is a stack that: (a) grows towards low addresses, and (b) the stack pointer points to the next available memory position**, *i.e.*, the memory position that must be used to store the next item to be pushed into the stack. Pushing a value into an empty descending stack is performed by first storing the value into the memory word pointed by the stack pointer (an empty memory word), then decreasing the stack pointer. Popping a value from a full descending stack is performed by first increasing the stack pointer then loading the value from the memory word pointed by the stack pointer.

A **full ascending stack is a stack that: (a) grows towards high addresses, and (b) the stack pointer points to the last element pushed into the stack**. Pushing a value into a full ascending stack is performed by increasing the stack pointer and then storing the value into the memory word pointed by the stack pointer. Popping a value from a full ascending stack is performed by first loading the value from the memory word pointed by the stack pointer and then decreasing the stack pointer.

An **empty ascending stack is a stack that: (a) grows towards high addresses, and (b) the stack pointer points to the next available memory position**, *i.e.*, the memory position that must be used to store the next item to be pushed into the stack. Pushing a value into an empty ascending stack is performed by first storing the value into the memory word pointed by the stack pointer (an empty memory word), then increasing the stack pointer. Popping a value from a full ascending stack is performed by first decreasing the stack pointer then loading the value from the memory word pointed by the stack pointer.

NOTE: The RISC-V program stack is a full descending stack

.

8.3 The ABI and software composition

The Application Binary Interface, or ABI, is a set of conventions defined to facilitate the composition of software. The ABI defines, for example, the calling convention, which determines how and where parameters must be passed to routines and how and where values must be returned from routines.

To illustrate its importance, let us assume John implemented a routine called `jsort` that takes two arguments: a pointer to an array of characters and an integer that indicates the size of the array. The following code shows the signature of the `jsort` routine:

```
void jsort(char* a, int n);
```

Also, lets assume John made this routine available through a binary library.

Now, let us assume Mary has John's library and wants to invoke the `jsort` routine. So far, we know that Mary can link her program with John's library and invoke the `jsort` routine by executing a `jal jsort` instruction. However, where should Mary place the routine parameters, *i.e.*, the pointer to the array (`char* a`) and the size of the array (`int n`)?

The answer to the previous question is "it depends on where the code implemented by John is expecting the parameters". For example, if the `jsort` routine is expecting the first parameter (the pointer to the array) to be placed at register `a0` and the second one (the array size) to be placed at register `a1`, then, Mary has to place these parameters in these two registers, otherwise, John's code will not work properly.

The calling convention, defined by the ABI, defines where routine parameters must be passed. Assuming John and Mary are following the same ABI, it should be easy for Mary to place the routine parameters in the correct registers.

There may be multiple, different, ABIs defined for a single computer architecture. This is the case for x86, for example, with different ABIs defined for different operating systems. The RISC-V consortium defines several ABIs[3]. Unless otherwise stated, in this text we will use the RISC-V `ilp32` ABI, which defines that `int`, `long`, and `pointers` are all 32-bits long. It also defines that `long long` is a 64-bit type, `char` is 8-bit, and `short` is 16-bit.

NOTE: Only code generated for the same ABI can be linked together by the linker.

NOTE: When generating code with GCC, the user may specify the ABI using the `-mabi` flag. For example, the following command may be used to compile and assemble the `program.c` file using the `ilp32` ABI: `gcc -c program.c -mabi=ilp32 -o program.o`

8.4 Passing parameters to and returning values from routines

8.4.1 Passing parameters to routines

Some routines have parameters, hence, when invoking them, the program needs to pass these parameters. Routine parameters can be passed through registers or the

[3]As of August 2020, there are two integer ABIs and three floating-point ABIs for RISC-V

program stack. As discussed in Section 8.3, the proper places to pass parameters to a routine is defined by the ABI.

The RISC-V `ilp32` ABI defines a set of conventions to pass parameters to routines[4]. These conventions specify how different types of values (char, integer, structs, ...) must be passed on registers or the stack. To simplify the discussion, we will focus on scalar values that can be represented with 32 or fewer bits. These values can be (unsigned) `char`, (unsigned) `short`, (unsigned) `integer`, or `pointer values`. The integer calling convention specifies the following rules to pass these type of values as parameters:

- The first eight scalar parameters of the routine are passed through registers `a0` to `a7`, being one parameter per register. Hence, if the routine has less than 9 parameters, all of them are passed through registers. Integer scalars narrower than 32 bits (*e.g.*, char or short) are widened according to the sign of their type up to 32 bits, then sign-extended to 32 bits.

- In case there are more than 8 parameters, the remaining parameters, *i.e.*, parameters 9 to N, are passed through the stack. In this case, the parameters must be pushed into the program stack. The last parameter, *i.e.*, the N^{th} parameter, must be pushed first and the 9^{th} one must be pushed last. These parameters must be later removed from the stack by the same routine that pushed them into the stack. Again, integer scalars narrower than 32 bits (*e.g.*, char or short) are widened according to the sign of their type up to 32 bits, then sign-extended to 32 bits. Consequently, scalar values smaller than 32 bits are expanded to 32 bits and occupy 4 bytes when passed as parameters through the program stack.

Before invoking a routine, the caller must set the parameters, *i.e.*, they must be placed in the registers and into the stack accordingly to the ABI. To illustrate how to pass parameters, let us assume there is a routine called `sum10` that takes, as parameters, 10 integer values, sums them, and returns the result. The following code shows the `sum10` signature:

```
1 int sum10(int a, int b, int c, int d, int e,
2           int f, int g, int h, int i, int j);
```

According to the RISC-V `ilp32` ABI, to invoke this routine, one must place parameters `a`, `b`, `c`, `d`, `e`, `f`, `g`, and `h` on registers `a0`, `a1`, `a2`, `a3`, `a4`, `a5`, `a6`, and `a7`, respectively. Also, parameters `i` and `j` must be placed on the stack, being the value of `j` pushed first and the value of `i` last.

The following code shows how to call the `sum10` routine passing as arguments values `10`, `20`, `30`, `40`, `50`, `60`, `70`, `80`, `90`, `100`. Notice that the 9^{th} and the 10^{th} parameters are pushed into the stack (lines 11-15). Also, notice that these parameters are later removed from the program stack by the same routine that placed them on the program stack (line 17), *i.e.*, the `main` routine.

```
1  # sum10(10,20,30,40,50,60,70,80,90,100);
2  main:
3   li a0, 10        # 1st parameter
4   li a1, 20        # 2nd parameter
5   li a2, 30        # 3rd parameter
6   li a3, 40        # 4th parameter
7   li a4, 50        # 5th parameter
8   li a5, 60        # 6th parameter
9   li a6, 70        # 7th parameter
10  li a7, 80        # 8th parameter
11  addi sp, sp, -8  # Allocate stack space
```

[4] https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md

```
12 li t1, 100       # Push the 10th parameter
13 sw t1, 4(sp)
14 li t1, 90        # Push the 9th parameter
15 sw t1, 0(sp)
16 jal sum10        # Invoke sum10
17 addi sp, sp, 8   # Deallocate the parameters from stack
18 ret
```

The routine that was invoked (`sum10` in this example) must retrieve the parameters from the registers and the stack accordingly to the ABI. The following code shows a possible implementation for the `sum10` routine and illustrates this process. It loads the values of the 9^{th} and the 10^{th} parameters from the stack into registers `t1` and `t2`. Notice that, even though it loads these values from the program stack, it does not pop (deallocate) the values from the program stack - this cleaning process must be performed by the caller routine, *i.e.*, the routine that invoked the `sum10` routine.

```
1  sum10:
2  lw  t1, 0(sp)   # Loads the 9th parameter into t1
3  lw  t2, 4(sp)   # Loads the 10th parameter into t2
4  add a0, a0, a1  # Sums all parameters
5  add a0, a0, a2
6  add a0, a0, a3
7  add a0, a0, a4
8  add a0, a0, a5
9  add a0, a0, a6
10 add a0, a0, a7
11 add a0, a0, t1
12 add a0, a0, t2  # Place return value on a0
13 ret             # Returns
```

8.4.2 Returning values from routines

The RISC-V `ilp32` ABI defines that values should be returned in register `a0`. In case the value being returned is 64-bit long, then the least significant 32 bits must be returned in register `a0` and the most significant 32 bits must be returned in register `a1`.

8.5 Value and reference parameters

Parameters may be passed as values or as references to variables. **Value parameters are parameters that contain the value itself**, *i.e.*, the value is placed directly into the registers or the program stack. The following code shows a routine that expects its parameter to be passed by value. This routine takes a value as a parameter, computes its power of two, and returns the result.

```
1 int pow2(int v)
2 {
3     return v*v;
4 }
```

Accordingly to the RISC-V `ilp32` ABI, this parameter must be passed in register `a0`. Since it is passed as value, register `a0` will contain the value itself. The following code shows the implementation of the previous "C" code in assembly language. Notice that the code multiplies the contents of register `a0` by itself.

```
pow2:
  mul a0, a0, a0 # a0 = a0 * a0
  ret            # return
```

The following code shows how to invoke the `pow2` routine to compute the square of 32. Notice that the value itself is directly placed into register `a0`.

```
main:
  li  a0, 32  # set the parameter with value 32
  jal pow2    # invoke pow2
  ret
```

Reference parameters are parameters that contain a reference to a variable. This "reference" is the variable memory address. Hence, the routine may use this address to read or update the variable value. The following code shows a routine that expects its parameter to be passed by reference. This routine updates the variable value by increasing its contents by one.

```
void inc(int* v)
{
    *v = *v + 1;
}
```

Again, accordingly to the RISC-V `ilp32` ABI, this parameter must be passed in register `a0`. Since it is passed as reference, register `a0` will contain the address of the variable. The following code shows the implementation of the previous "C" code in assembly language. Notice that the code uses the address passed in register `a0` to update the contents of the variable.

```
inc:
  lw   a1, (a0)  # a1 = *v
  addi a1, a1, 1 # a1 = a1 + 1
  sw   a1, (a0)  # *v = a1
  ret
```

The following code shows how the `inc` routine can be invoked to increase the value of variable `y`. Notice that the address of variable `y`, instead of its value, is loaded into register `a0`.

```
.data
y: .skip 4

.text
main:
  la   a0, y  # set the parameter with the address of y
  jal  inc    # invoke inc
  ret
```

Reference parameters can be used to pass information in and out of routines. Since a reference is essentially a memory address, the information being passed into or out of the routine must be located in the memory.

8.6 Global vs local variables

In high-level languages, such as "C", global variables are variables declared outside routines, and they can be accessed on any routine in the program.

The following code shows an example of a "C" program with a global variable named `x`. Notice that the global variable may be accessed from within any routine.

```
1 int x;
2
3 int main()
4 {
5     return x+1;
6 }
```

Global variables are allocated on the static data space by the assembler and are usually declared on assembly programs with the help of directives. The following code shows the assembly code for the previous "C" program.

```
1 .data
2 x:
3     .skip 4
4
5 .text
6 main:
7     la   a0, x      # Loads the address of variable x
8     lw   a0, 0(a0)  # Loads the value o x
9     addi a0, a0, 1  # Increments the value
10        ret             # Return
```

In the previous code, the `.data` directive (line 1) informs the assembler that the following contents must be placed into the static data space. The `x` label marks the address of variable `x`. The `.skip 4` directive (line 3) instructs the assembler to skip four bytes, which is used to allocate space for variable `x`. The `.text` directive (line 5) informs the assembler that the following contents must now be placed into the code space. The remainder of the code (lines 6-10) implements the main routine.

In high-level languages, such as in "C", local variables are variables declared inside routines, and can be used only inside the routine that declared it.

Ideally, local variables should be allocated on registers. The following code contains a local variable called `tmp` that can be allocated on a register.

```
1 void exchange(int* a, int* b)
2 {
3     int tmp = *b;
4     *b = *a
5     *a = tmp;
6 }
```

The following code shows the assembly code for the previous "C" code. Notice that the local variable `tmp` was allocated on register a2.

```
1 exchange:
2   lw a2, (a1)   # tmp = *b
3   lw a3, (a0)   # a3 = *a
4   sw a3, (a1)   # *b = a3
5   sw a2, (a0)   # *a = tmp
6   ret
```

8.6.1 Allocating local variables on memory

There are several situations in which local variables must be allocated on memory, for example:

- When a routine has a large number of local variables and there are not enough registers to allocate them; or
- When a local variable is an array or a structure; or
- When the code needs the address of the local variable. This may be the case when passing a local variable as reference to other routines.

Local variables that need to be allocated on memory, are allocated on the program stack whenever a routine is invoked and become active, and deallocated whenever the routine returns, *i.e.*, it is not active anymore. These variables must be allocated and deallocated by the routine that contains them. They must be allocated upon the entry of a routine and deallocated before returning from the routine.

Allocating space at the program stack is performed by decreasing the value of the stack pointer by the number of bytes that need to be allocated. The following code shows an example in which a local variable named `userid` needs to be allocated on memory. The address of this variable, which is a 4-byte integer, is passed to routines `get_uid`.

```
1 int foo()
2 {
3     int userid;
4     get_uid(&userid);
5     return userid;
6 }
```

The following code shows the assembly code for the `foo` routine. First, the stack pointer is decreased to allocate space for the `userid` variable (line 2). Then, the address of the `userid` variable is loaded into register `a0` to be passed as parameter to routine `get_uid` (line 3). Notice that, since the last element added to the program stack was the `userid` variable, the stack pointer points to (contains the address of) this variable. Next, the `get_uid` routine is invoked (line 4) and, after returning, the value of the `userid` variable is loaded into register `a0` to be returned (line 5)[5]. Finally, before returning from the `foo` routine, the stack pointer is increased to deallocate the `userid` variable from the program stack.

```
1 foo:
2   addi sp, sp, -4    # Allocate userid
3   mv   a0, sp        # a0 = address of userid (&userid)
4   jal  get_uid       # Invoke the get_uid routine
5   lw   a0, (sp)      # a0 = userid
6   addi sp, sp, 4     # Deallocate userid
7   ret
```

The following code shows another example in which a local variable needs to be allocated in memory. In this case, the `my_array` local variable needs to be allocated on memory because it is an array. Also, the address of this variable is passed to routine `init_array`.

```
1 int bar()
2 {
```

[5]Since the value of variable `userid` may have been modified by the `get_uid` routine, the code needs to load the value of variable `userid` from memory after the execution of the `get_uid` routine.

```
3     int my_array[8];
4     init_array(my_array);
5     return my_array[4];
6 }
```

The following code shows the assembly code for the `bar` routine. First, the stack pointer is decreased to allocate space for the `my_array` variable (line 2)[6]. Then, the address of the `my_array` variable is loaded into register `a0` to be passed as parameter to routine `init_array` (line 3). Again, since the last element added to the program stack was the `my_array` variable, the stack pointer points to (contains the address) of this variable[7]. Next, the `init_array` routine is invoked (line 4) and, after returning, the value of `my_array[4]` is loaded into register `a0` for return (line 5)[8]. Finally, before returning from the `bar` routine, the stack pointer is increased to deallocate the `my_array` variable (line 6) from the program stack.

```
1 bar:
2   addi sp, sp, -32   # Allocate my_array
3   mv   a0, sp        # a0 = address of my_array
4   jal  init_array    # Invoke the init_array routine
5   lw   a0, 16(sp)    # Load my_array[4] into a0
6   addi sp, sp, 32    # Deallocate my_array
7   ret
```

The following code shows yet another example in which a local variable needs to be allocated in memory. In this case, the `d` local variable needs to be allocated on memory because it is a struct. Also, the address of this variable is passed to routine `init_date`.

```
1 typedef struct
2 {
3   int year;
4   int month;
5   int day;
6 } date_t;
7
8 int get_current_day()
9 {
10    date_t d;
11    init_date(&d);
12    return d.day;
13 }
```

The following code shows the assembly code for the `get_current_day` routine. First, the stack pointer is decreased to allocate space for the `d` variable (line 2)[9]. Then, the address of variable `d` variable is loaded into register `a0` to be passed as parameter to routine `init_date` (line 3). Again, since the last element added to the program stack was variable `d`, the stack pointer points to (contains the address) of this variable[10]. Next, the `init_date` routine is invoked (line 4) and, after returning, the value of `d.day` is loaded into register `a0` for return (line 5)[11]. Finally, before returning

[6] Notice that the `my_array` variable is a 32-byte long array - It contains eight 4-byte integers.

[7] At this point, the stack pointer points to the first element of the `my_array` array, *i.e.*, `my_array[0]`.

[8] Since the stack pointer (`sp`) is pointing to `my_array[0]`) and each array element has four bytes, `my_array[4]` is located at memory position SP+16.

[9] Notice that the `d` variable is a 12-byte long struct - It contains three 4-byte integers.

[10] At this point, the stack pointer points to the first element of the `data_t` struct, *i.e.*, the `year` field.

[11] Since the stack pointer (`sp`) is pointing to `d.year`) and each field has four bytes, `d.day` is located at memory position SP+8.

from the `get_current_day` routine, the stack pointer is increased to deallocate the `d` variable (line 6) from the program stack.

```
1 get_current_day:
2   addi sp, sp, -12   # Allocate d
3   mv   a0, sp        # a0 = address of d
4   jal  init_date     # Invoke the init_date routine
5   lw   a0, 8(sp)     # Load d.day into a0
6   addi sp, sp, 12    # Deallocate d
7   ret
```

8.7 Register usage policies

The assembly code examples in the previous sections have used registers to hold variables and temporary values, to return values from routines, and to pass parameters to routines. In fact, registers are frequently used resources. Before using a register, however, it may be necessary to save its contents to memory so that it can be restored later.

Let us use as an example the `exchange` routine, introduced in Section 8.6. The following codes show its implementation in assembly language.

```
1 exchange:
2   lw a2, (a1)   # tmp = *b
3   lw a3, (a0)   # a3 = *a
4   sw a3, (a1)   # *b = a3
5   sw a2, (a0)   # *a = tmp
6   ret
```

Notice that the assembly code uses registers `a2` and `a3` to perform the computation. In this case, the contents of these two registers are destroyed by the `lw` instructions (lines 2 and 3).

Now, lets assume that the `mix` routine loads an "important information" on register `a2` and, before using this information, it invokes the `exchange` routine. The following code illustrates this situation. First, the `mix` routine loads the important information into register `a2` (line 2). Then, it sets the parameters and invoke the `exchange` routine (lines 3-5). Finally, the `mix` routine returns the important information by copying it from register `a2` to register `a0` (line 6) and executing the `ret` instruction (line 7).

```
1 mix:
2   lw  a2, (a0)   # load important information into a2
3   la  a0, x      # Sets parameter 0 with address of var. x
4   la  a1, y      # Sets parameter 1 with address of var. y
5   jal exchange   # Invokes exchange to swap x an y values
6   mv  a0, a2     # Move important information into a0 to return
7   ret
```

Notice, however, that the `exchange` routine destroys the contents of registers `a2`. Consequently, the value returned by the `mix` routine is not the "important information" that was loaded into register `a2` at line 2.

To solve the problem, the `mix` routine could save the contents of register `a2` on the program stack before invoking the `exchange` routine and restore it after the `exchange` routine returns. The following code illustrates this situation. Notice that the contents of register `a2` are saved into the program stack (lines 3 and 4) before invoking the `exchange` routine and restored (lines 8 and 9) after the `exchange` routine returns.

```
1  mix:
2    lw   a2, (a0)   # load important information into a2
3    addi sp, sp, -4 # Saves a2: Allocate stack space
4    sw   a2, (sp)   #           Store a2 into the stack
5    la   a0, x      # Sets parameter 1 with address of var. x
6    la   a1, y      # Sets parameter 1 with address of var. y
7    jal  exchange   # Invokes exchange to swap x an y values
8    lw   a2, (sp)   # Restores a2: Loads a2 from the stack
9    addi sp, sp,  4 #              Deallocate the stack space
10   mv   a0, a2     # Move important information into a0 to return
11   ret
```

Another way to solve this problem is to modify the `exchange` routine to save and restore all the registers that it might change. The following code illustrates this situation. Notice that the contents of registers `a2` and `a3` are saved into the program stack (lines 2-4) at the beginning of the routine and restored (lines 9-11) before returning from the routine.

```
1  exchange:
2    addi sp, sp, -8 # Allocate stack space
3    sw   a2, 4(sp)  # Save contents of a2
4    sw   a3, 0(sp)  # Save contents of a3
5    lw   a2, (a1)   # tmp = *b
6    lw   a3, (a0)   # a3 = *a
7    sw   a3, (a1)   # *b = a3
8    sw   a2, (a0)   # *a = tmp
9    lw   a3, 0(sp)  # Restore contents of a3
10   lw   a2, 4(sp)  # Restore contents of a2
11   addi sp, sp, 8  # Deallocate stack space
12   ret
```

8.7.1 Caller-saved vs Callee-saved registers

A call site is a place from which a routine is being invoked. In the previous example, the `jal exchange` instruction defines a call site. For each call site, there is a caller and a callee routine. **The caller routine is the routine that is invoking the other routine**, *i.e.*, the routine that contains the call site. **The callee routine is the routine that is being invoked by the call site**. In the previous example, the callee is the `exchange` routine and the caller is the `mix` routine.

The previous examples discussed two alternative solutions to preserving the contents of register `a2` across a call site: the contents are saved and restored by the caller routine or the contents are saved and restored by the callee routine. Even though both approaches are correct, there should be a convention so that programmers (and compilers) do not need to inspect the code of the callee routine to figure out whether or not to save the contents of a register before invoking the callee.

The ABI defines which registers must be saved by the caller, *i.e.*, the routine that is invoking, and which registers must be saved by the callee routine, *i.e.*, the routine that is being invoked. **Caller-saved registers are registers that must be saved by the caller routine** and **callee-saved registers are registers that must be saved by the callee routine**.

The RISC-V `ilp32` ABI defines that **registers `t0-t6`, `a0-a7`, and `ra` are caller-saved**. Also, it defines that **registers `s0-s11` are callee-saved**.

According to the RISC-V `ilp32` ABI, the contents of register `a2` must be preserved by the caller routine, hence, in previous example, the `mix` routine is responsible for saving the contents of register `a2` before invoking the `exchange` routine.

It is important to notice that there is no need to save all the caller-saved registers before invoking a routine. Only the registers that contain values that might be used

the caller routine after the call site. In the previous example, the `mix` routine must save `a2` because it needs the value of `a2` after the call site.

Also, it is important to notice that there is no need for the callee routine to save and restore all the callee-saved registers, only the ones that are modified by the routine. As an example, there is no need for the `exchange` routine to save registers `s0-s11` since it does not modify these registers.

8.7.2 Saving and restoring the return address

As discussed before, whenever a routine is invoked, the return address is stored at the return address register (`ra`). In other words, whenever a routine is invoked, register `ra` is updated with a new value, and its previous contents are destroyed. Consequently, if the contents of register `ra` are required after the call site, then it must be saved and restored. This is usually the case because the code that is invoking a routine usually belongs to another routine, hence, it might need the return address to return its execution to the caller – Notice that the `ret` pseudo-instruction reads the contents of register `ra` to return the execution flow to the correct place.

Since register `ra` is a caller-save register, the caller routine must save its contents. In the previous example, the `mix` routine must have saved and restored the contents of `ra` to prevent it from being destroyed when invoking the `exchange` routine. The following code shows the correct code for the `mix` routine.

```
1  mix:
2    addi sp, sp, -4 # Saves ra: Allocate stack space
3    sw   ra, (sp)   #           Store ra into the stack
4    lw   a2, (a0)   # load important information into a2
5    addi sp, sp, -4 # Saves a2: Allocate stack space
6    sw   a2, (sp)   #           Store a2 into the stack
7    la   a0, x      # Sets parameter 1 with address of var. x
8    la   a1, y      # Sets parameter 1 with address of var. y
9    jal  exchange   # Invokes exchange to swap x an y values
10   lw   a2, (sp)   # Restores a2: Loads a2 from the stack
11   addi sp, sp,  4 #              Deallocate the stack space
12   mv   a0, a2     # Move important information into a0 to return
13   lw   ra, (sp)   # Restores ra: Loads ra from the stack
14   addi sp, sp,  4 #              Deallocate the stack space
15   ret
```

Leaf routines are routines that do not call other routines. Since they do not call other routines, the contents of register `ra` are not modified. Hence, there is no need to save the return address on the stack when implementing leaf routines. In the previous examples, the `exchange` routine is a leaf routine, hence, there is no need to save and restore the contents of the return address.

Finally, the standard ABI specifies that routines should not modify the integer registers `tp` and `gp`, because signal handlers may rely upon their values.

8.8 Stack Frames and the Frame Pointer

8.8.1 Stack Frames

All active routines may contain information in the program stack. Also, this information is naturally grouped and sorted accordingly to the call order. For example, lets assume that routine `A` invoked routine `B`, routine `B` invoked routine `C`, and routine `C` is currently being executed. Notice that routines `A` and `B` are still active. The contents added by routine `A` on the stack are placed before the contents of routine `B`. Also, the contents added by routine `B` on the stack are placed before the contents of routine `C`. Figure 8.3 illustrates the state of the program stack when routine `C` is executing.

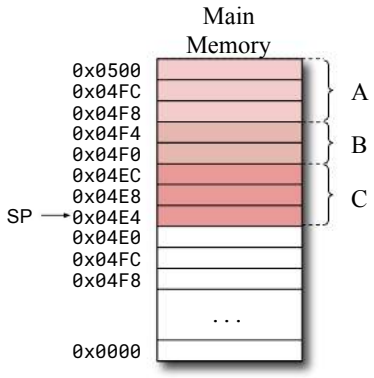


Figure 8.3: Program stack with data from three active routines `A`, `B`, and `C`.

A stack frame is a consecutive segment of data on the program stack that stores information about an active routine. In the previous example, there are three stack frames when routine `C` is executing. The first one, comprised by addresses `0x0500` and `0x04F8`, is the stack frame for routine A[12].

8.8.2 The Frame Pointer

The previous sections discussed several situations in which the program stack is used to store information about active routines. Whenever new information is added to the program stack, the stack pointer moves and this must be accounted for when accessing information that was pushed before. For example, lets analyze the `addijx` routine, which takes ten arguments, invokes the `get_x` routine, adds its return value to the 9th (`i`) and the 10th (`j`) parameters and returns the result.

```
1 int addijx(int a, int b, int c, int d, int e,
2            int f, int g, int h, int i, int j)
3 {
4     return get_x() + i + j;
5 }
```

The following code shows the `addijx` routine implemented in assembly. Notice that the return address is saved to (lines 2 and 3) and restored from (lines 9 and 10) the program stack. At the entry point, the stack pointer points to the 9th parameter (`i`), however, after the return address is saved on (pushed into) the stack, the stack pointer points to the return address. Hence, to access the 9th parameter after this point, the code must add four to the stack pointer (line 5).

```
1  addijx:
2    addi sp, sp, -4 # Saves the
3    sw   ra, (sp)   #  return address
4    jal  get_x      # Invoke the get_x routine
5    lw   a1, 4(sp)  # Loads i from the program stack
6    lw   a2, 8(sp)  # Loads j from the program stack
7    add  a0, a1, a1 # a0 = get_x() + i
8    add  a0, a2, a2 # a0 = get_x() + i + j
9    lw   ra, (sp)   # Restore the
10   addi sp, sp, 4  #  return address
11   ret             # Returns
```

[12]Notice that the same routine may be invoked multiple times before returning, hence, this routine may have multiple activations and, hence, multiple stack frames. This is usually the case of recursive routines.

The more information is added to the stack, the harder it may get to track the addresses of all parameters and local variables across the routine. One way of mitigating this problem is to keep a fixed pointer to the stack so that all parameters and local variables can be accessed using this pointer plus a fixed offset. **The frame pointer points to the beginning of the stack frame of the currently executing routine**. As a consequence, it provides a fixed pointer to the stack across the execution of a routine and may be used as a fixed reference to access parameters and local variables.

In the RISC-V `ilp32` ABI, the frame pointer is stored by the frame pointer register, or `fp`. The `fp` register must be initialized at the beginning of the routine, however, its previous contents must be saved so that it can be restored before returning from the routine. Also, in most cases, instead of pushing information one by one on the program stack, each stack frame can be allocated with a single instruction at the beginning of the routine and deallocated with a single instruction before returning. The following code shows an example in which the stack frame is allocated (deallocate) in the beginning (end) of the routine (lines 2 and 15) and the frame pointer is used to access the parameters using a fixed offset (lines 8 and 9).

```
1  addijx:
2    addi sp, sp, -8 # Allocates the stack frame
3    sw   ra, 4(sp)  # Saves return address
4    sw   fp, 0(sp)  # Saves previous frame pointer
5    addi fp, sp, 8  # Adjust frame pointer.
6
7    jal  get_x      # Invoke the get_x routine
8    lw   a1,  (fp)  # Loads i from the program stack
9    lw   a2, 4(fp)  # Loads j from the program stack
10   add  a0, a1, a1 # a0 = get_x() + i
11   add  a0, a2, a2 # a0 = get_x() + i + j
12
13   lw   fp, 0(sp)  # Restore previous frame pointer
14   lw   ra, 4(sp)  # Restore return address
15   addi sp, sp, 8  # Deallocate the stack frame
16   ret             # Returns
```

In the previous example, the `addijx` stack frame had 8 bytes and stored the return address and the previous frame pointer. In case more registers need to be saved or local variables need to be stored on the program stack, the stack frame may be easily increased by changing the constant (8) in lines 2 and 15.

8.8.3 Keeping the stack pointer aligned

The RISC-V `ilp32` ABI specifies that the stack pointer shall always be aligned to a 128-bit boundary upon routine entry. Also, the documentation states that "In the standard ABI, the stack pointer must remain aligned throughout procedure execution. Non-standard ABI code must realign the stack pointer prior to invoking standard ABI procedures."

One way of ensuring that the stack pointer is always aligned throughout the routine execution is to always increase and decrease it by multiples of 16 since 16 bytes is equal to 128 bits. In this context, the programmer (or the compiler) can always allocate stack frames using multiples of 16 bytes.

8.9 Implementing RISC-V `ilp32` compatible routines

The following list provides a set of guidelines to help programmers implement RISC-V assembly routines compatible with the RISC-V `ilp32` ABI.

- Include a label to define the routine entry point. When translating "C" code to assembly code, the label must match the "C" routine name;
- Use the return instruction (`ret`) to return from the routine. This instruction jumps to the address that is stored in the return address register (`ra`);
- Parameters must be accessed accordingly to the RISC-V `ilp32` ABI. Considering scalar parameters smaller than or equal to 32 bits, the first eight parameters are expected in registers `a0` to `a7`, and the remaining ones on the stack. Also, integer scalars narrower than 32 bits (*e.g.*, char or short) are widened according to the sign of their type up to 32 bits;

  Parameters passed on the stack are organized so that, the last parameter, *i.e.*, the N^{th} parameter, is pushed first and the 9^{th} is pushed last. As a consequence, upon the routine entrance, the stack pointer points to the 9^{th} parameter, `sp`+4 points to the 10^{th} parameter, and so on. Parameters are allocated on the program stack by the caller routine and must also be deallocated by the caller routine. The callee must not deallocate parameters allocated by the caller;
- In case the routine needs to store information on the program stack, a stack frame should be allocated at the beginning of the routine and deallocated before returning. The size of the stack frame must be a multiple of 16 to ensure the stack pointer keeps aligned to a 128-bit boundary, as required by the standard ABI;
- The routine may use registers to implement its functionality, however, callee-saved registers that are modified by the routine must be saved in the beginning of the routine and restored before returning from it. These registers must be saved on the stack frame;
- The routine may modify and use caller-saved registers without saving them, however, in case their value needs to be preserved across a call site, the routine must save (restore) its contents before (after) the call site. Routines that call other routines must save and restore the return address register to preserve its contents across call sites. These registers must be saved on the stack frame;
- Local variables may be allocated on registers or on memory. Local variables that need to be allocated on memory must be allocated on the stack frame;
- Optionally, the frame pointer register (`fp`) may be used to keep a pointer to the beginning of the stack frame and provide a fixed reference to access parameters and local variables. In this case, the previous frame pointer must be preserved when returning from the routine, hence, the contents of the frame pointer register must be saved in the stack frame at the beginning of the function and restored before returning.
- The standard ABI specifies that routines should not modify the integer registers `tp` and `gp`.

8.10 Examples

This section presents examples of assembly code generated for "C" routines.

8.10.1 Recursive routines

Recursive routines are routines that call themselves. The following code shows an example of a recursive routine that computes the factorial of a number.

```
1 int factorial(int n)
2 {
3   if (n>1)
```

```
4     return n * factorial(n-1);
5   else
6     return 1;
7 }
```

Notice that, if the parameter `n` is greater than one, then factorial of `n` is computed by multiplying the value of `n` by the factorial of `n-1`, which is computed by a recursive call to the `factorial` routine.

Generating code for a recursive routine is as simple as generating code for any non-leaf routine. The following code shows how the previous recursive routine can be implemented in assembly. First, the stack frame is allocated and the return address is saved on it (lines 2 and 3). Then, `n` is compared with `1` (lines 4 and 5) and, if `n` is less or equal to one, the code jumps to the "else block" (lines 12 and 13), otherwise, it proceeds to the "then block" (lines 6 to 11). The "else block" simply sets `a0` with 1 for return and proceeds with the routine finalization code, *i.e.*, the code that restores the return address, deallocates the stack frame and returns (lines 15 to 17). The "then block" (lines 6 to 11) implements the code that performs the recursive call (*i.e.*, `n * factorial(n-1)`). First, it saves the value of `a0` (`n`) on the stack frame to preserve it across the call site (line 6). Then, it sets the parameter for the recursive call and invokes the routine[13] (lines 7 and 8). Next, it recovers the value of `n` from the stack frame into register `a1` (line 9) and multiplies it by the value returned by the recursive call, which is located in `a0` (line 10). Finally, it jumps to the `fact_end` label to execute the routine finalization code.

```
1  factorial:
2    addi sp, sp, -16   # Allocates the routine frame
3    sw   ra, 0(sp)     # Saves the return address
4    li   a1, 1
5    ble  a0, a1, else  # if (n>1)
6    sw   a0, 4(sp)     # Saves n (a0) on the routine frame
7    addi a0, a0, -1    # Set the parameter (n-1)
8    jal  factorial     # Perform the recursive call
9    lw   a1, 4(sp)     # Loads n from the routine frame (into a1)
10   mul  a0, a0, a1    # a0 = factorial(n-1) * n
11   j    fact_end      # Jumps to end
12  else:
13   li   a0, 1         # Set the return value to 1
14  fact_end:
15   lw   ra, 0(sp)     # Restores the return addres
16   addi sp, sp, 16    # Deallocate the routine frame
17   ret                # Return
```

8.10.2 The standard "C" library syscall routines

As discussed in Section 6.7.4, user programs usually invoke operating system service routines to perform input and output operations. This operation, called syscall, is performed in RISC-V by executing the `ecall` instruction. In this case, the program must set the `a7` register with the proper syscall code. The standard "C" library provides routines to help users invoke the syscalls. The `write` routine is one of these routines and has the following signature:

```
1 ssize_t write(int fildes, const void *buf, size_t nbyte);
```

This routine takes three parameters: the file descriptor (`fildes`), a pointer to the buffer that contains the information that must be written to the file (`buf`), and the

[13] The only difference between a recursive routine and a non-leaf regular routine, is that the recursive one is invoking the same routine while other on-leaf routines invoke other routines.

number of bytes that must be written (`nbyte`). Also, it returns the number of bytes written to the file.

The following assembly code shows a possible implementation for the `write` routine. This routine receives parameters `fildes`, `buf`, and `nbyte` on registers `a0`, `a1`, and `a2`, respectively. These parameters are the same parameters that must be passed to the syscall and are already placed on the correct registers, hence, there is no need to adjust the parameters when invoking the system call on line 5.

```
1 write:
2   addi sp, sp, -16  # Allocates the stack frame
3   sw   ra, 12(sp)   # Saves the return address
4   li   a7, 64       # Sets the syscall code (64 = write)
5   ecall             # Invokes the operating system
6   lw   ra, 12(sp)   # Restores the return address
7   addi sp, sp, 16   # Deallocates the stack frame
8   ret               # Returns
```

Part III

System-level programming

Chapter 9

Accessing peripherals

As discussed in the previous chapters, the CPU executes programs that are stored on the main memory. In this process, the CPU fetches the program's instructions from the main memory and executes them, which may cause the CPU to load or store data on the main memory. The previous chapters also explain that user-level programs perform input and output operations by invoking the operating system.

This chapter discusses how programs may directly interact with input and output hardware devices to perform input and output operations. This task is useful when developing software for a system that does not contain an operating system or when implementing operating systems' components, such as device drivers.

The remainder of the chapter is organized as follows: Section 9.1 introduces the concept of peripherals and discusses how they are connected to the CPU. Section 9.2 presents the two main methods for programs to interact with peripherals: port-mapped I/O and memory-mapped I/O. Section 9.3 discusses how I/O operations are performed on RISC-V-based computing systems. Finally, Section 9.3 discusses the busy waiting concept.

9.1 Peripherals

Peripherals are input/output, or I/O, devices that are connected to the computer. There are several kinds of peripherals. Mouse, keyboard, image scanners, barcode readers, game controllers, microphones, webcams, and read-only memories are examples of input devices. Monitors, projectors, printers, headphones, and computer speakers are examples of output devices. There are also devices that perform both input and output operations, such as data storage devices (including a disk drive, USB flash drive, memory card, and tape drive), network cards, *etc.*

Input and output devices interface with the CPU through a bus, which is a communication system that transfers information between the computer components. This system is usually composed of wires that are responsible for transmitting the information and associated circuitries, which orchestrate communication. Figure 9.1 illustrates a computer system in which a system bus connects the CPU, the main memory, a persistent storage device (HDD), an input device, and an output device.

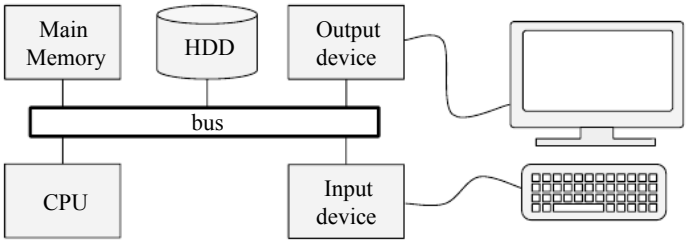


Figure 9.1: Computer system components connected through a system bus.

Peripherals usually contain registers or an internal memory that are accessed by

the CPU to perform input and output operations. To discuss this concept, let us consider a hypothetical computing system that has a seven-segment display (an output peripheral) attached to a display controller, which, in turn, is connected to the CPU through the bus, as illustrated in Figure 9.2.

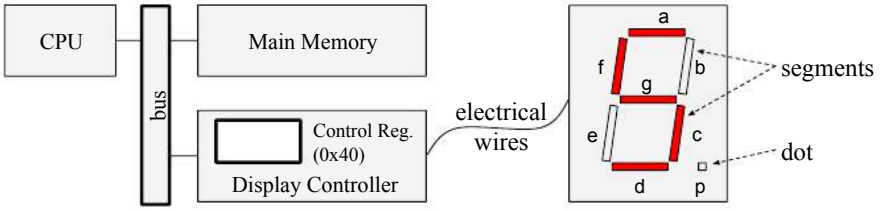


Figure 9.2: Computing system with a seven-segment display and a display controller.

Seven-segment displays are devices that contain seven segments and one dot that can be light up individually. Modern seven-segment displays are implemented using one light-emitting diode (LED) per segment and one for the dot. The segments and the dot are positioned on the display so that it is possible to display patterns that resemble decimal digits by lighting up a subset of the display segments. For example, one may turn on segments a, f, g, c, and d to show a pattern that resembles the decimal digit '5', as illustrated in Figure 9.2.

The display controller is the device responsible for controlling the seven-segment display. It is connected to the seven-segment display LEDs using electrical wires, and it turns on or off each one of the segments and the dot according to the contents of an eight-bit register called control register. Each bit of the control register (Control Reg.) controls whether each display segment or dot must be turned on or off. In this case, bits 7, 6, 5, 4, 3, 2, 1, and 0 (the rightmost one) control the dot (p), and the segments a, b, c, d, e, f, and g, respectively. Figure 9.3 shows the value that must be written into the control register (`0x5b`) to turn on the segments that show a pattern that resembles the decimal digit '5'.

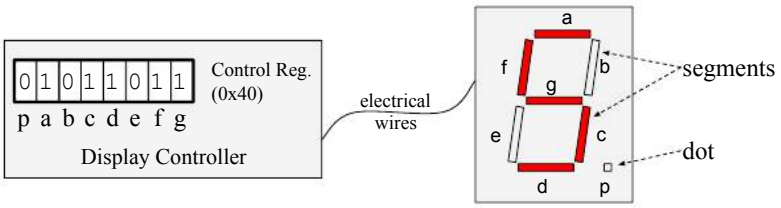


Figure 9.3: A detailed view of the display controller control register.

In the previous example, the seven-segment display is controlled by a display controller, which, in turn, turns on and off the display segments and the dot accordingly to the contents of the control register. In this context, to perform an output operation using the display controller, the CPU must write a value into the display controller control register. In general, to perform an output or input operation using an I/O device, the CPU writes or reads from a register, or internal memory, that belongs to the input device controller.

The next section discusses how the CPU may write/read to/from peripheral registers and their internal memories.

9.2 Interacting with peripherals

The CPU interacts with the main memory by sending addresses, data, and commands through the bus. For example, to write the 8-bit value `0x55` into the memory word associated with address `0x8000`, a CPU may send the `WRITE` command, the address, and the value to the main memory through the bus. Alternatively, to read data from a memory word, the CPU may send the `READ` command and the memory word

address and wait for the main memory to place the data on the bus so it can copy the data into one of its internal registers. The CPU employs the same process to interact with peripherals' controllers, *i.e.*, the CPU sends/receives information (commands, addresses, and data) to/from controllers through the bus.

CPUs are usually connected physically to the main memory and to peripherals' controllers through one or more buses. There are several kinds of buses and their organization and implementation may vary dramatically. For example, some buses may employ a single set of wires to transmit addresses, data, and commands, while others may use dedicated wires for each one of these tasks. Also, the number of buses and their disposition on the system may vary significantly across computing systems.

Even though buses' implementation and organization may vary dramatically, their characteristics are usually transparent to the programmer, *i.e.*, they do not affect how the programmer generates code that interacts with peripherals' controllers nor the main memory. The CPU ISA usually provides the programmer with instructions that hide the details (*e.g.*, their protocols and inner workings) of how the CPU or the peripherals interact with each other or the bus. These instructions allow the programmer to instruct the CPU to write/read data to/from peripherals' registers and their internal memories in a simple way. For example, the RV32I ISA contains load and store instructions (*e.g.*, `lw` and `sw`[1]) that allow the programmer to instruct the CPU to read/write data from/to the main memory without worrying how the bus that connects the CPU to the main memory works.

There may be several peripherals on the system, and each one of them may have multiple registers or internal memories. Hence, there must be a way for programmers to specify the proper peripheral register or internal memory position to be accessed by the instruction. This is usually performed by associating each peripheral register and internal memory position with a different identifier, often an integer number, which may be known as an address or an I/O port. In this context, instructions used for interacting with peripherals usually identify the peripheral register or memory position by its address or I/O port.

Sections 9.2.1 and 9.2.2 discuss the two main methods of accessing peripheral registers and their internal memories by executing CPU instructions.

9.2.1 Port-mapped I/O

Port-mapped I/O, also known as isolated I/O, is a method of accessing peripheral's registers and their internal memories that employ special ISA instructions for I/O operations. There are two central concepts in this method: I/O ports and I/O instructions. **An I/O instruction is a special instruction dedicated to access peripherals**. In contrast, **an I/O port is an unsigned integer number that identifies peripherals' registers and internal memory words**. The programmer uses this identifier to specify which peripheral register or internal memory words must be accessed when performing an I/O operation with an I/O instruction.

To illustrate this concept, let us consider the IA-32 ISA family[2]. These ISAs contain two I/O instructions that copy data between CPU registers and a peripheral register or internal memory word: "input from port" [2] and "output to port" [3].

The input from port instruction, or `in`, takes two operands, the I/O port and a destination CPU register, `al`, `ax`, or `eax`[3]. The instruction copies a value from a peripheral register, or internal memory word, identified by the I/O port operand, into the destination CPU register. The following code shows an example in which the `in` instruction is used to read an 8-bit value from I/O port `0x71` and place it on the `al` CPU register.

```
1 in 0x71, %al
```

[1] See Section 6.6 for more information on RV32I load and store instructions.

[2] The IA-32 ISA family is a set of ISAs developed by Intel and based on the ISA used on the 8086 microprocessor.

[3] `al`, `ax`, and `eax` are 8-bit, 16-bit, and 32-bit CPU registers on the IA-32 ISA.

The output to port instruction, or `out`, also takes two operands; however, the first one specifies the target I/O port and the second one the source CPU register. The `out` instruction copies the value from the source CPU register into the peripheral register, or internal memory word, identified by the I/O port operand. The following code shows an example in which the `out` instruction is used to write the 8-bit value stored at the CPU register `al` into the peripheral register (or internal memory word) identified by the I/O port `0x70`.

```
1 out %al, 0x70
```

9.2.2 Memory-mapped I/O

The I/O address space defines the set of valid I/O port values. For example, in IA-32 ISAs, the I/O address space consists of 2^{16} (64 KB) individually addressable 8-bit I/O ports, numbered `0` through `0xFFFF` [1]. In the Port-mapped I/O method, the I/O address space is distinct from the main memory address space. In other words, a main memory word may be associated with a memory address (*e.g.*, `0x70`) that has the same value of an I/O port (*e.g.*, `0x70`) that is mapped to a peripheral register.

Memory-mapped I/O is a method of accessing peripheral's registers and their internal memories that employ regular memory access instructions. In the memory-mapped I/O method, there is a single address space, and some subsets of this space are mapped to main memory words while others are mapped to peripheral registers and internal memory words. In this context, the same instructions that read/write data from/to the main memory (*e.g.*, load and store instructions) are used to read/write data from/to peripheral registers and their internal memories. The address is the information that defines whether a main-memory word or a peripheral register or internal memory is accessed. Figure 9.4 shows an address space mapped to the main memory and multiple peripherals on a real computing system[4]. Addresses `0x70000000` to `0x80000000` are mapped to main memory words; hence, load and store operations on these addresses cause the CPU to read and store data on the main memory. On the other hand, performing a load/store operation on address `0x53F84000` causes the CPU to read/write data from/to a register on the GPIO peripheral.

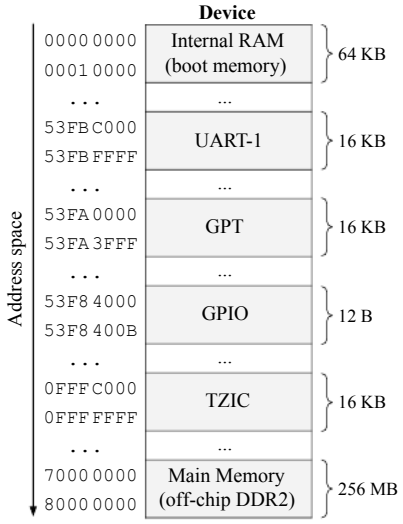


Figure 9.4: A single address space mapped to main memory (addresses `0x70000000` to `0x80000000`) and multiple peripherals.

[4]This is the address mapping employed on the Freescale i.MX53 platform. The UART (Universal asynchronous receiver-transmitter), the GPT (General Purpose Timer), the GPIO (General-Purpose Input/Output), and the TZIC (Trusted-Zone Interrupt Controller) are peripherals in the system.

9.3 I/O operations on RISC-V

Input and output operations on RISC-V ISAs, including the RV32I ISA, are performed using the memory-mapped I/O method. Hence, input operations are performed by executing load instructions (*e.g.*, `lw`) while output operations are performed with store instructions (*e.g.*, `sw`) on addresses that are mapped to peripheral registers or internal memory words.

To illustrate I/O operations on RISC-V, let us consider an elevator computing system that contains an RV32I CPU, the main memory, a seven-segment display, and a floor sensor, as shown in Figure 9.5.

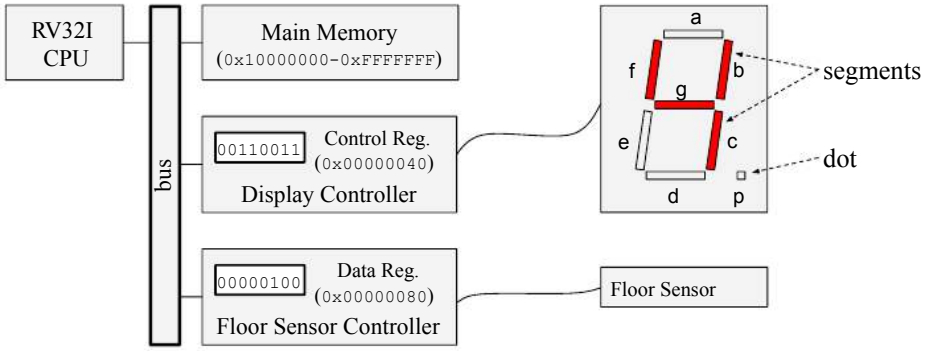


Figure 9.5: RV32I-based computing system with a seven-segment display and a floor sensor.

The seven-segment display is controlled by a display controller, as discussed in previous sections. Nonetheless, in this example, the control register is mapped to address `0x00000040`. The floor sensor detects on which floor the elevator is located while the floor sensor controller registers this information on the data register (Data Reg.), an 8-bit register situated in the floor sensor controller, and mapped to address `0x00000080`. For example, if the elevator is located on the fourth floor, the floor sensor controller stores the value four (`0b00000100`) on the data register.

The following code shows a routine that reads the elevator floor from the floor sensor controller (lines 6 and 7), translates the floor number into a configuration byte (lines 8 to 10), and writes the configuration byte into the display controller control register to set the seven-segment display (lines 11 and 12).

```
1  .section .text
2  .set DISPLAY_CONTROL_REG_PORT, 0x00000040
3  .set FLOOR_DATA_REG_PORT, 0x00000080
4
5  update_display:
6    li  a0, FLOOR_DATA_REG_PORT      # Reads the floor number and
7    lb  a1, (a0)                     #  store into a1
8    la  a0, floor_to_pattern_table   # Converts the floor number
9    add t0, a0, a1                   #  into a configuration
10   lb  a1, (t0)                     #  byte
11   li  a0, DISPLAY_CONTROL_REG_PORT # Sets the display controller
12   sb  a1, (a0)                     #  with the configuration byte
13   ret                              # Returns
14
15 .section .rodata
16 floor_to_pattern_table:
17   .byte 0x7e,0x30,0x6d,0x79,0x33,0x5b,0x5f,0x70,0x7f,0x7b
```

As discussed in Section 9.1, each bit of the configuration byte controls whether each segment (or the dot) is turned on or off. In this context, the code must convert

the floor number to a configuration byte that turns on a subset of the segments so the pattern displayed resembles the floor number. For example, if the elevator is located on the fourth floor (floor number = 4), the code must write the value `0x33` (`0b00110011`) to turn on segments b, c, f, and g, as illustrated in Figure 9.5. Notice that the code employs a table (`floor_to_pattern_table`) that can be indexed by the floor number to retrieve the proper configuration byte.

9.4 Busy waiting

Busy waiting is a technique in which the code waits for some condition to become true by repeatedly checking the condition on a loop. Once the condition becomes true, the code leaves the loop and proceeds with the rest of the execution.

In many situations, when interacting with peripherals, the program may need to wait for some condition to become true before performing an output or input operation. In these cases, the programmer may employ the busy waiting technique to control the execution flow. To illustrate this concept, let us consider a computing system that contains an RV32I CPU, the main memory, and a keypad, as shown in Figure 9.6.

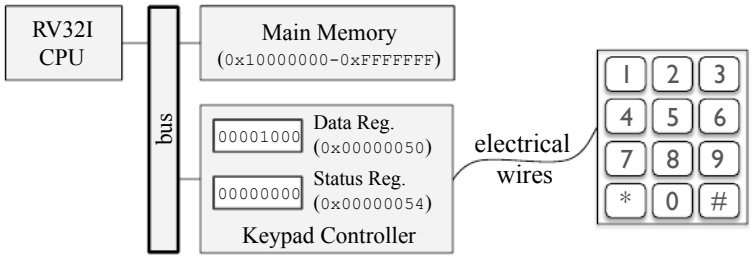


Figure 9.6: RV32I-based computing system with a keypad.

The keypad is connected to a keypad controller, which, in turn, is connected to the CPU through the bus. The keypad controller contains two 8-bit registers: the data register and the status register. The data register, which is mapped to address `0x00000050`, stores a value that indicates the last key pressed on the keypad. For example, if the keypad key '8' is pressed, then, the data register is set with number 8[5].

The status register, which is mapped to address `0x00000054`, contains a value that indicates the keypad's current status. The least significant bit of the status register (bit 0), called the READY bit, indicates whether the keypad was pressed since the last time the CPU read a value from the data register. If it contains the value 0, it means no keys were pressed since the last time the CPU read a value from the data register. If it contains the value 1, at least one key was pressed since the last time the CPU read a value from the data register. The second least significant bit of the status register (bit 1), called the OVRN bit, indicates whether the keypad was pressed more than once since the last time the CPU read a value from the data register. If it contains the value 1, it means the keypad was pressed more than once since the last time the CPU read a value from the data register. Notice that, since the keypad controller contains only one data register, and it stores the value of the last key pressed, if the keypad is pressed more than once before the CPU gets the chance to read the data register, one or more key values will be lost. This situation, known as overrun (OVRN), may be detected by inspecting the status register's OVRN bit.

Now, let us suppose the programmer needs to implement a routine called `read_keypad` that waits until the keypad is pressed and then returns the value of the last key pressed. This routine also needs to return the value -1 if the keypad was pressed more than once since a value was read from the keypad controller's data register. The

[5]Keys '#' and '*' produces the numbers 10 and 11.

following code shows an implementation of the `read_keypad` routine that employs the busy waiting technique to wait for the keypad to be pressed before reading the keypad controller's data register. First, it reads the contents of the keypad status register into register `a0` (lines 8 and 9) and then it checks whether the READY bit is set by performing a bit-wise and operation with the mask defined by the `READY_MASK` symbol (line 10) and jumping back to the beginning of the routine in case the result is zero (line 11). Next, it checks if the keypad was pressed more than once by performing a bit-wise and operation with the mask defined by the `OVRN_MASK` symbol (line 12) and jumping to the `ovrn_occured` label in case the result is not zero (line 13). Finally, it reads the key value from the keypad controller's data register (lines 14 and 15) and returns.

```
1  .text
2  .set  DATA_REG_PORT, 0x00000050
3  .set  STAT_REG_PORT, 0x00000054
4  .set  READY_MASK,    0b00000001
5  .set  OVRN_MASK,     0b00000010
6
7  read_keypad:
8    li   a0, STAT_REG_PORT  # Reads the keypad
9    lb   a0, 0(a0)          #  status into a0
10   andi t0, a0, READY_MASK # Check the READY bit and
11   beqz t0, read_keypad    #  until it is equal to 1
12   andi t0, a0, OVRN_MASK  # Check if OVRN bit and jump
13   bnez t0, ovrn_occured   #  to ovrn_occured if equals to 1
14   la   a0, DATA_REG_PORT  # Reads the key from the
15   lb   a0, 0(a0)          #  data register into a0
16   ret                     # Return
17 ovrn_occured:
18   li   a0, -1             # Returns -1
19   ret                     # Return
```

Chapter 10

External Interrupts

10.1 Introduction

As discussed in previous chapters, the CPU fetches and executes instructions from the main memory. In this context, most of the actions that happen in the system are initiated by the CPU, as a result of executing instructions. For example, reading/writing data to/from the main memory and to/from peripherals are events triggered by the CPU when executing instructions. However, there are some events that are initiated by other hardware components, such as peripherals. For example, in the system discussed in Section 9.4, when a keypad key is pressed, the keypad controller registers this information on the keypad controller registers. Even though the events were not initiated by the CPU, it might require the CPU attention, *i.e.*, it might require the CPU to perform some action. Hence, there must be a way to inform the CPU that the peripheral needs its attention.

To illustrate this concept, let us consider the computing system depicted in Figure 10.1, which contains a RV32I CPU, the main memory, and a keypad.

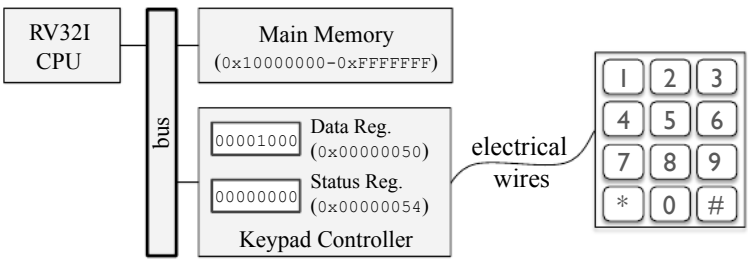


Figure 10.1: RV32I-based computing system with a keypad.

The keypad is connected to a keypad controller, which, in turn, contains two 8-bit registers: the data register and the status register. The data register, which is mapped to address `0x00000050`, stores a value that indicates the last key pressed on the keypad. The status register, which is mapped to address `0x00000054`, contains a value that indicates the keypad's current status. The least significant bit of the status register (bit 0), called the READY bit, indicates whether the keypad was pressed since the last time the CPU read a value from the data register. If it contains the value 0, it means no keys were pressed since the last time the CPU read a value from the data register, otherwise, it contains the value 1. The second least significant bit of the status register (bit 1), called the OVRN bit, indicates whether the keypad was pressed more than once since the last time the CPU read a value from the data register. If it contains the value 1, it means the keypad was pressed more than once since the last time the CPU read a value from the data register. Since the keypad controller contains only one data register, if the keypad is pressed more than once before the CPU gets the chance to read the data register, one or more key values are lost. This situation, known as data overrun (OVRN), may be detected by inspecting the status register's OVRN bit.

The longer the program takes to read the data register contents, the higher is the chance of a data overrun. To prevent data overruns, it is customary to copy the data register value to a first-in first-out (FIFO) queue[1] located at the main memory as soon as the keypad is pressed. This approach is illustrated in Figure 10.2, which implements the FIFO queue using an 8-element circular buffer and two pointers, one that points to the queue head (oldest element inserted) and another that points to the queue tail (last element inserted). In this example, the keypad keys '1', '9', and '6', have been pressed and stored on the queue.

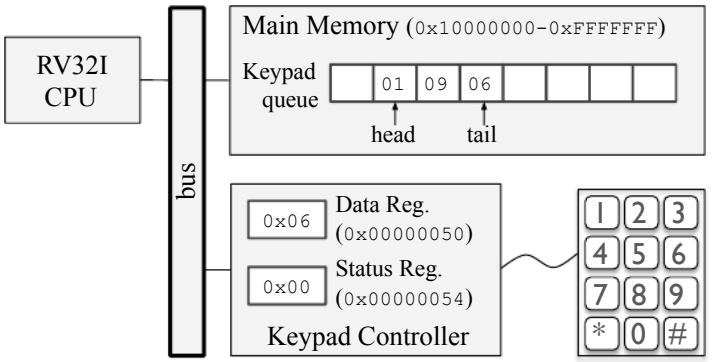


Figure 10.2: Storing the contents of the data register on a queue located at the main memory.

In this approach, whenever a key is pressed, its value is pushed into the queue's tail, and whenever the user program needs to read a key, it pops it from the queue's head, instead of reading from the keypad data register. Notice that the queue works as a buffer that is capable of storing multiple key values, allowing the program to perform longer computations before reading each key value. Figure 10.3 illustrates what happens when the keypad key '9' is pressed. First, the key '9' is pressed ❶. Then, the keypad controller registers this information on the data and the status registers ❷. Finally, the CPU executes a routine that pushes the data register value on the queue's tail ❸.

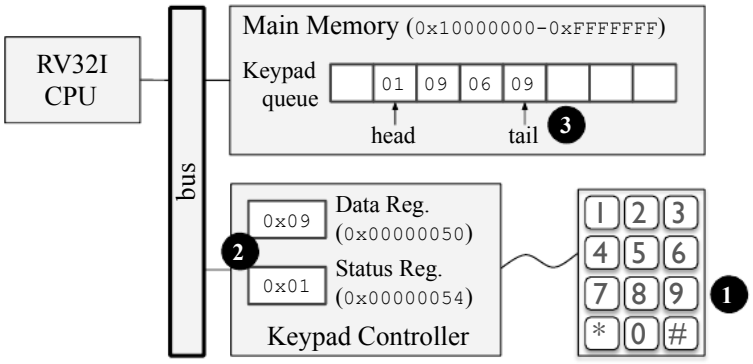


Figure 10.3: Storing the key value on the queue when the keypad key '9' is pressed.

Copying the value from the keypad's data register to the queue located at the main memory is usually performed by the CPU, through the execution of a routine. In this context, whenever the keypad is pressed, the CPU must execute this routine as soon as possible to prevent data overruns. There are two main methods to direct the CPU attention to handle events caused by external hardware: Polling and Hardware Interrupts.

[1]Fixed-length queues can be efficiently implemented on memory using circular buffers.

10.1.1 Polling

Polling is a method in which the program is designed so that the CPU periodically checks whether peripherals need attention. In this approach, the program has to be designed so that it checks the peripherals that may need CPU attention from time to time. For example, the program may contain a main loop that repeatedly checks the peripherals and perform some computation. Whenever there is a peripheral that needs attention, the program invokes a routine to handle the peripheral. Algorithm 3 illustrates a program that employs polling to handle peripherals. It is composed of a main loop (the outer while loop) that checks peripherals for attention and perform some computation alternatively.

Algorithm 3: Handling peripherals with polling.

```
1 while True do
2 |   // Handle peripherals
3 |   for p in Peripherals do
4 |   |   if needsAttention(p) then
5 |   |   |   handlePeripheral(p) ;
6 |   |   end
7 |   end
8 |   PerformSomeComputation();
9 end
```

Algorithm 4 illustrates a code that employs polling to check and handle the keypad periodically. In this case, the *keypadPressed()* function checks whether the keypad READY bit is set, if so, then it returns true and the program invokes the *getKey()* and the *pushKeyOnQueue()* routines to read the contents of the data register and push it to the queue's tail. The *Compute()* routine represents the work that is done by the program in the meantime.

Algorithm 4: Handling the keypad with polling.

```
1 while True do
2 |   if keypadPressed() then
3 |   |   k ← getKey() ;
4 |   |   pushKeyOnQueue(k) ;
5 |   end
6 |   Compute() ;
7 end
```

Notice that the amount of work performed by the *Compute()* routine affects the frequency in which the keypad is checked. On the one hand, the longer the *Compute()* routine takes to execute, the higher is the chance of occurring data overrun. On the other hand, breaking the computation so that each call to *Compute()* executes quickly (*e.g.*, performing just a small fraction of the computation every time it is invoked) may cause a large overhead (checking peripherals may take a long time) and may make the program hard to design and implement. As a consequence, polling is usually not the best approach to check for and handle peripherals events.

10.2 External Interrupts

Hardware Interrupts is a mechanism that allows hardware to inform the CPU they require attention. **External Interrupts are interrupts caused by external (non-CPU) hardware, such as peripherals, to inform the CPU they require attention**. In this approach, the peripheral sends an interrupt signal to the CPU, and, once the CPU receives this signal, it:

1. saves the context[2] of the current program;

[2]The context is defined by the program values, which are stored at CPU registers and the main memory.

2. invokes a routine to handle the hardware interrupt;

3. restores the context of the saved program and continues executing.

To illustrate this concept, let us consider the computing system depicted in Figure 10.4.

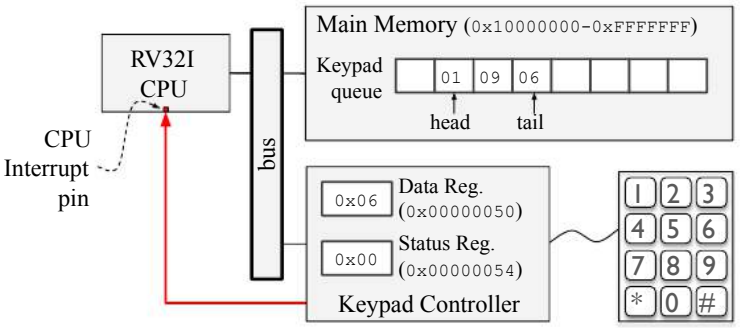


Figure 10.4: RV32I-based computing system with a keypad connected to the CPU interrupt pin.

This system is very similar to the one presented in Figures 10.1, 10.2, and 10.3. The main difference is that the CPU contains an interrupt pin and the keypad controller is connected to the CPU interrupt pin (red arrow). **The interrupt pin is an input pin that informs the CPU whether or not there is an external interrupt**. In this context, whenever a key is pressed, the keypad controller sends a signal to the CPU through the interrupt pin. The CPU hardware (i) constantly monitors the interrupt pin and, in case it receives an interrupt signal, it interrupts the current execution flow to execute an interrupt service routine. **The interrupt service routine[3], or ISR, is a software routine that handles the interrupt**. There are several ways of implementing ISRs, however, in general, they usually save the context of the executing program (*e.g.*, the contents of the CPU registers) on main memory, interact with the peripheral that sent the interrupt signal, and, finally, restore the saved context so that the CPU continues executing the program that was interrupted.

10.2.1 Detecting external interrupts

Algorithm 5 illustrates how the CPU instruction execution cycle presented at Section 1.2 may be adapted to detect external interrupts. In this example, before fetching an instruction for execution, it verifies if the `interrupt_pin` is set, *i.e.*, if the CPU received an interrupt signal, and if interrupts are enabled, *i.e.*, if the `interrupts_enabled` is set. If both conditions are met, it saves the contents of the program counter (PC) into the `SAVED_PC` register, sets the PC register with the address of the interrupt service routine (`ISR_ADDRESS`), and disables interrupts by clearing the `interrupts_enabled` register. As a result, the next instruction that will be fetched

[3]The interrupt service routine is also known as interrupt handler.

for execution is the first instruction of the interrupt service routine.

Algorithm 5: Adapting the CPU instructions execution cycle to handle interrupts.

```
1  while True do
2  |  // Check for interrupts
3  |  if (interrupt_pin = '1') and (interrupts_enabled = '1') then
4  |  |  // Invoke the ISR
5  |  |  SAVED_PC ← PC ;
6  |  |  PC ← ISR_ADDRESS;
7  |  |  interrupts_enabled ← '0';
8  |  end
9  |  // Fetch instruction and update PC
10 |  IR ← MainMemory[PC] ;
11 |  PC ← PC+4;
12 |  ExecuteInstruction(IR);
13 end
```

Notice that the pseudo-code in Algorithm 5 disables interruptions whenever it invokes an interrupt service routine. This is performed so the interrupt service routine has a chance to save all the important context (including the contents of the `SAVED_PC`) before the CPU redirects the execution flow to handle a new interrupt. Moreover, in some systems, the interrupt service routine is responsible for interacting with the peripheral so it stops signaling the interrupt pin. In these cases, the CPU must ignore the `interrupt_pin` until the peripheral stops signaling the `interrupt_pin`. Once it is safe to handle new interrupts, the interrupt service routine may set the `interrupts_enabled` register so the CPU may handle new interrupts.

> **NOTE:** CPUs usually disable interrupts on power-up to allow the boot software to configure the hardware and register the proper ISRs before the system tries to handle interruptions.

10.2.2 Invoking the proper interrupt service routine

A computing system may contain several peripherals that may interrupt the CPU. Also, each peripheral usually requires a specialized routine to handle its interrupts. For example, a keypad controller may require a routine that adds the value from the data register into a queue on the main memory while a pointing device (mouse) controller may need another action. As a consequence, whenever an interrupt occurs, the system has to perform two tasks: (i) identify which peripheral interrupted the CPU, and (ii) invoke the proper routine to handle the interrupt.

Depending on the system architecture, these two tasks may be performed by hardware, by software, or by a combination of both. In fact, there are several ways of identifying which peripheral interrupted the CPU and invoking the proper routine to handle the interrupt. To discuss the main trade-offs, we will consider three distinct designs that we will call: SW-only, SW/HW, and HW-only.

SW-only design

In the SW-only design, the ISR is responsible for identifying which peripheral interrupted the CPU, and invoking the proper routine to handle the interrupt. In this approach, upon an interrupt, the CPU invokes a generic ISR that must perform both tasks. Since there is no hardware support to identify which peripheral interrupted the CPU, the ISR may have to interact with all peripherals to find out which one is requiring the CPU attention. Once the ISR finds out which peripheral interrupted the CPU, it can invoke the proper interrupt service routine to handle the peripheral interrupt.

The main advantage of this approach is that it simplifies the CPU hardware design, which is usually an important goal since hardware bugs are hard to find and do not

allow easy patching once the CPU is manufactured and sold. Nonetheless, in case there are several peripherals or peripherals are slow, the ISR may take a long time trying to figure out which peripheral interrupted the CPU. This may affect overall system performance and may even cause the system to lose data due to data overruns, as discussed in previous sections.

SW/HW design

In the SW-HW design, the ISR is also responsible for performing both tasks; however, the hardware provides some support to identify which peripheral interrupted the CPU. In this case, upon an interrupt, the hardware sets a register[4] with a value that indicates which peripheral generated the interrupt. Consequently, the ISR may simply read this register to find out which peripheral generated the interrupt. Once the ISR finds out which peripheral interrupted the CPU, it can invoke the proper interrupt service routine to handle the peripheral interrupt.

Both the SW-only and the SW/HW designed jumps to a single generic ISR. This approach is known as "direct mode" in RISV-V terminology. Algorithm 5 illustrates how this approach may be implemented in hardware.

The CPU hardware design may not be as simple as in the SW-only approach; however, in this approach, the ISR takes very little time (usually the time required to execute one or two instructions) to figure out which peripheral send the interrupt signal.

HW-only design

In the HW-only design, the hardware is responsible for identifying which peripheral interrupted the CPU and to invoke the proper ISR. In this case, each peripheral is associated with an interrupt identifier[5] and the CPU must automatically map this identifier to its respective ISR. This is usually performed with a table, often called interrupt vector table, that maps the interrupt identifier to the address of the ISR[6].

To illustrate this concept, let us consider a system in which each peripheral is associated with a unique interrupt identifier that may range from 0 to 15, and that the CPU automatically registers the interrupt identifier on the `INTERRUPT_ID` register whenever an interrupt signal is received.Also, there is an array on main memory, called interrupt vector table, that contains in position i the address of the ISR that must be invoked to handle interrupts from the peripheral that is associated with interrupt identifier i. The system also contains a register called `INT_TABLE_BASE` that stores the interrupt vector table base address. In this context, to invoke the proper ISR, the CPU may load the ISR address from the interrupt vector table using the interrupt identifier. Algorithm 6 illustrates how a CPU may automatically load the address of the proper ISR by accessing the interrupt vector table. The CPU multiplies the contents of the `INTERRUPT_ID` register by four because each entry in the interrupt

[4]This register may be an internal CPU register or a register on an interrupt controller, which is a peripheral designed to support external interrupt handling.

[5]In some systems this identifier is called Interrupt Request, or IRQ.

[6]Some designs map the interrupt identifier to the first instruction of the ISR, which is usually a jump to the routine.

vector table contains a 32-bit (four-byte) address.

Algorithm 6: Adapting the CPU instructions execution cycle to automatically invoke the proper ISR.

```
1  while True do
2  |  // Check for interrupts
3  |  if (interrupt_pin = '1') and (interrupts_enabled = '1') then
4  |  |  // Save the previous PC
5  |  |  SAVED_PC ← PC ;
6  |  |  // Retrieve the ISR address from the interrupt vector table and set PC
7  |  |  PC ← MainMemory[INT_TABLE_BASE + INTERRUPT_ID × 4];
8  |  |  interrupts_enabled ← '0' ;
9  |  end
10 |  // Fetch instruction and update PC
11 |  IR ← MainMemory[PC] ;
12 |  PC ← PC+4;
13 |  ExecuteInstruction(IR);
14 end
```

On power-up, before enabling interrupts, the boot software must write the interrupt vector table on main memory and set the `INT_TABLE_BASE` register with its base address.

This approach's main advantage is the performance since the CPU directly invokes the proper ISR upon an interrupt. However, the CPU hardware design usually becomes more complicated.

10.3 Interrupts on RV32I

In this section, we will discuss external interrupts in the context of RISV-V CPUs. As we will discuss in Section 11.1, the RISC-V Instruction Set Architecture defines three privilege levels: User/Application, Supervisor, and Machine. Also, it specifies that microprocessor may implement only a subset of these privilege levels. To simplify the discussion, in this chapter we will focus on systems that implement only the Machine privilege levels, which is usually the case of embedded systems. Chapter 11 will discuss other privilege levels and how they affect the RISC-V interrupt handling mechanism.

10.3.1 Control and Status Registers

The RISC-V Control and Status Registers, or CSRs, are special registers that expose the CPU status to the software and allow it to configure the CPU behavior. For example, on the RV32I ISA, the `mstatus` CSR is a 32-bit register that contains several bits that expose the current status of the CPU or control the CPU behavior.

The RISC-V ISA contains a set of special instructions to enable software to inspect and modify the contents of CSRs [4]. The `csrrw rd, csr, rs1` instruction atomically swaps values in the CSRs and integer registers. For example, `csrrw a0, mscratch, a0` atomically swaps the contents of register `a0` and the `mscratch` CSR. The `csrr rd, csr` instruction copies the contents of the `csr` CSR into the `rd` general-purpose register. For example, the `csrr a0, mstatus` copies the contents of the `mstatus` CSR into `a0`. The `csrw csr, rd` instruction copies the contents of the `rd` general-purpose register into the `csr` CSR. For example, the `csrw mtvec, a1` copies the contents of register `a1` into the `mtvec` CSR.

The RV32I Control and Status Registers are 32-bit long and some of them may contain subfields with different purposes. For example, the `mstatus` CSR, illustrated in Figure 10.5, contains more than 17 subfields (*e.g.*, MIE, MPIE, *etc.*), each one with a specific purpose.

In our discussion, we will use the `csr.FIELD` notation to refer to the `FIELD` subfield of `csr` CSR. For example, `mstatus.MIE` refers to the `MIE` subfield present on the

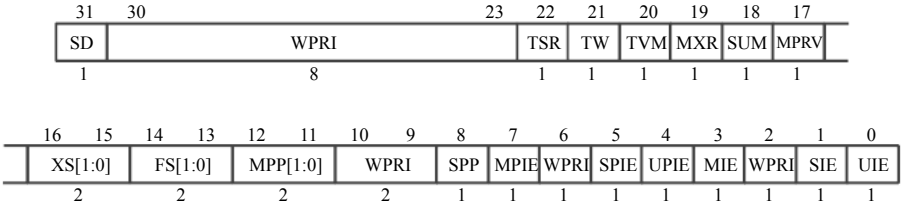


Figure 10.5: The `mstatus` CSR and its subfields.

`mstatus` CSR.

10.3.2 Interrupt related Control and Status Registers

The following RV32I Control and Status Registers contain status or may control the behavior of the interrupt handling mechanism on Machine level:

- `mstatus`: The following `mstatus` subfields provide information or control the interrupt handling mechanism.
  - `mstatus.MIE` (Machine Interrupt Enable): The machine interrupt enable subfield controls whether or not the CPU will handle interrupts. If it contains zero, the CPU ignores all interrupts. This bit is cleared upon reset and the boot software usually sets it with one once peripherals and the interrupt handling mechanism are properly configured;

    > **NOTE:** There is also a CSR called Machine Interrupt Enable; however, it has a different purpose and should not e confused with the `mstatus.MIE` subfield.

  - `mstatus.MPIE` (Machine Previous Interrupt Enable): Upon an interrupt, the CPU changes the value of the `mstatus.MIE` subfield to ignore new interrupts until the interrupt service routine enables it again. The previous value of the `mstatus.MIE` subfield is automatically saved by the CPU on the `mstatus.MPIE` subfield;
  - `mstatus.MPP`: Upon an interrupt, the CPU may change the privilege mode. In this case, it saves the previous current privilege mode on this subfield.
- `mcause` (Machine Interrupt Cause): The machine interrupt cause CSR stores the interrupt cause, *i.e.*, a value that identifies why an interrupt was generated. It has two fields: `mcause.EXCCODE` (bits 0 to 30) and `mcause.INTERRUPT` (bit 31). The `mcause.INTERRUPT` subfield specifies whether the interrupt is an actual interrupt (`1`) or an exception (`0`)[7]. The `mcause.EXCCODE` subfield specifies the interrupt (or exception) identifier. On Machine mode, interrupts caused by peripherals are classified as "Machine external interrupt" and the value registered on the `mcause.INTERRUPT` and `mcause.EXCCODE` subfields are `0x1` and `0xB`, respectively.
- `mtvec` (Machine Trap Vector): The machine trap vector CSR stores information that allows the CPU to identify the proper interrupt service routine address when an interrupt occurs. It has two fields: `mtvec.MODE` (bits 0 to 1) and `mtvec.BASE` (bits 2 to 31). The `mtvec.MODE` specifies whether the CPU is working in the direct (`00`) of vectored (`01`) mode. In the direct mode, upon an interrupt, the CPU sets the PC with the contents of the `mtvec.BASE` subfield. In the vectored mode, upon an interrupt, the CPU sets the PC with `mtvec.BASE` + (4 × `mcause.EXCCODE`).

[7] In RISC-V terminology, exceptions are interrupts caused by the CPU. We will discuss these kind of interrupts on Chapter 11.

- `mie` (Machine Interrupt Enable): There may be several sources of interrupts on RISC-V CPUs. The machine interrupt enable CSR allows the system to configure which interrupts must be enabled or disabled, *i.e.*, which interrupts must be taken by the CPU and which must be ignored. The following subfields control the interrupt handling mechanism on the Machine mode:
  - `mie.MEIE`: The Machine External Interrupt Enabled subfield (bit 11) controls whether the CPU must accept or ignore external interrupts.
  - `mie.MTIE`: RISC-V CPUs contain an internal timer that may be configured to generate interrupts. The Machine Timer Interrupt Enabled subfield (bit 7) controls whether the CPU must accept or ignore interrupts from this timer.
  - `mie.MSIE`: The Machine Software Interrupt Enabled subfield (bit 3) controls whether the CPU must accept or ignore software interrupts[8] on Machine mode.

> **NOTE:** The `mstatus.MIE` subfield is implements a global interrupt enable mechanism while the `mie` CSR allows for a more fine-grained configuration. The CPU ignores all interrupts if the `mstatus.MIE` subfield contains the value `0`, even if the `mie` CSR is enabling interrupts.

- `mip` (Machine Interrupt Pending): The machine interrupt pending CSR registers which interrupts are pending, *i.e.*, they have been signaled but not handled by the CPU yet. The following subfields indicate the status of pending interrupts on the Machine mode:
  - `mip.MEIP`: The Machine External Interrupt Pending subfield (bit 11) indicates whether an external interrupt is pending.
  - `mip.MTIP`: The Machine Timer Interrupt Pending subfield (bit 7) indicates whether a timer interrupt is pending.
  - `mip.MSIP`: The Machine Software Interrupt Pending subfield (bit 3) indicates whether a software interrupt is pending.
- `mepc` (Machine Exception Program Counter): Upon an interrupt, the CPU saves the contents of the PC register into the machine exception program counter CSR.
- `mscratch` (Machine Scratch): The machine scratch CSR is a scratch register that is visible in machine mode. Section 10.3.4 discusses how it can be used to support the implementation of interrupt service routines.

10.3.3 Interrupt Handling Flow

As discussed in Section 10.2, to handle external interrupts, the system must save the context of the current program, execute a routine to handle the hardware interrupt, and, finally, restore the saved context so the CPU continues executing the previous program.

In RISC-V CPUs, a subset of these tasks is performed by the CPU itself while the remainder must be performed by the interrupt service routine. For example, the program counter is saved by the CPU itself while other general-purpose registers must be saved by the interrupt service routine. This section discusses the actions that are performed automatically by the CPU while Section 10.3.4 discusses the actions that must be performed by the interrupt service routine.

Algorithm 7 illustrates how a RV32I CPU handles external interrupts. First, it verifies if the CPU must accept interrupts by checking the contents of the `mstatus.MIE`

[8] Software interrupts is a special kind of interrupt generated by the CPU itself. These interrupts will be discussed on Chapter 11.

subfield (line 2). If it is set ('1'), then it checks for interrupts (lines 3-20), otherwise, it ignores interrupts and proceeds with the normal instruction execution cycle (line 22). Assuming `mstatus.MIE` = '1', in case there is an external interrupt pending (`mip.MEIP` = '1') and external interrupts are enabled (`mie.MEIE` = '1') (line 4), then the CPU handles the interrupt (lines 5-19). When handling an interrupt, the CPU first saves the value of the `mstatus.MIE` subfield and clears it so that new interrupts are ignored[9] (lines 6 and 7). Then, the CPU saves the contents of the PC into the `mepc` CSR (line 8), and sets the `mcause` CSR (lines 10 and 11). Finally, it changes the PC register so it points to the first instruction of the interrupt service routine (lines 13-19).

Algorithm 7: RV32I CPU external interrupt handling flow.

```
1  while True do
2      if mstatus.MIE = '1' then
3          // Check for external interrupts
4          if (mip.MEIP = '1') and (mie.MEIE = '1') then
5              // Save part of the context and ignore new interrupts
6              mstatus.MPIE ← mstatus.MIE ;
7              mstatus.MIE = '0' ;
8              mepc ← PC ;
9              // Sets the interrupt cause
10             mcause.INTERRUPT ← '1' ;
11             mcause.EXCCODE ← '0xB' ;
12             // Change PC to execute the ISR
13             if mtvec.MODE = '0' then
14                 // Direct mode (0)
15                 PC ← mtvec.BASE ;
16             else
17                 // Vectored mode (1)
18                 PC ← mtvec.BASE + (4 × mcause.EXCCODE) ;
19             end
20         end
21     end
22     // Fetch instruction and update PC
23     IR ← MainMemory[PC] ;
24     PC ← PC+4;
25     ExecuteInstruction(IR);
26 end
```

10.3.4 Implementing an interrupt service routine

As indicated by Algorithm 7, the RISC-V CPU hardware already saves part of the current program context before redirecting the execution flow to the interrupt service routine. Notice that the contents of the `mstatus.MIE` subfield and the contents of the `PC` register were automatically saved on the `mstatus.MPIE` subfield and the `mepc` CSR, respectively. The interrupt service routine is responsible for saving the remaining of the context before handling the interrupt.

A program context is defined by the program values, which are stored at CPU registers and the main memory. The interrupt service routine usually saves the registers' values by copying them into the main memory. Values that are already in the main memory, however, are not copied. They are preserved by designing the interrupt service routine so that it does not touch the memory words that were being used by the program that was executing.

Any register that may be changed by the interrupt service routine must be saved. In some cases, only a subset of the context needs to be saved. In others, when executing sophisticated interrupt service routines, for example, it may be necessary to save all registers.

[9] New interrupts are ignored until the interrupt service routine sets this subfield again

There are several strategies that may be employed to save the registers' contents in the main memory. In our discussion, we will assume there is a dedicated stack for interrupt service routines. This stack, called here ISR stack, is allocated on main memory on a set of addresses that does not collide with the addresses used by other programs running on the system. In this way, whenever an interrupt occurs, the interrupt service routine can safely save the context of the currently executing program into the ISR stack.

To push values into the ISR stack, we must first make the `SP` register point to the top of the ISR stack. In the RV32I ISA, this task can be performed with help from the `mscratch` CSR. To do so, we first configure the system so that the `mscratch` CSR points to the top of the ISR stack on power-up. Then, at the beginning/end of the interrupt service routine, we exchange the value of `mscratch` and `SP` by executing the `csrrw` instruction. The following code illustrates this process. First (line 3), the ISR swaps the `sp` and the `mscratch` registers' contents so that the `sp` register points to the top of the ISR stack and the `mscratch` points to the top of the previous program stack. Then, the ISR allocates space on the ISR stack and saves all the necessary context (lines 4-7). After this, it identifies the interrupt source by inspecting the `mcause` CSR and invokes the specialized ISR to handle the interrupt (lines 9-11). Finally, the ISR restores the context by loading the registers' values from the ISR stack, swapping the `mscratch` and `sp` registers' contents, and executing the `mret` instruction.

```
1  main_isr:
2    # Saves the context
3    csrrw sp, mscratch, sp  # Exchange sp with mscratch
4    addi  sp, sp, -64       # Allocates space at the ISR stack
5    sw a0, 0(sp)            # Saves a0
6    sw a1, 4(sp)            # Saves a1
7    ...
8
9    # Handles the interrupt
10   csrr  a1, mcause        # Reads the interrupt cause and perform
11   ...                     # some action according to the cause.
12
13   # Restores the context
14   ...
15   lw a1, 4(sp)           # Restores a1
16   lw a0, 0(sp)           # Restores a0
17   addi  sp, sp, 64       # Deallocate space from the ISR stack
18   csrrw sp, mscratch, sp # Exchange sp with mscratch
19   mret                   # Returns from the interrupt
```

The `mret` instruction is a special instruction that recovers the context that was automatically saved by the CPU hardware. More specifically, it recovers the `mstatus.MIE` subfield contents by copying the value from `mstatus.MPIE` and the PC register's contents by copying the values from the `mepc` register[10].

10.3.5 Setting up the Interrupt Handling Mechanism

For the interrupt handling mechanism to work properly, a set of tasks must be performed. These tasks, usually performed on the boot process, are discussed in the following sections.

Registering the interrupt service routine(s)

To register the interrupt service routine, the system must write the address of the ISR (direct mode) or the base address of the interrupt vector table (vectored mode)

[10] In systems with multiple privilege modes (*e.g.*, Machine and User modes), the `mret` instruction also recovers the privilege mode.

on the mtvec CSR. Assuming the main_isr routine starts on an address that is a multiple of four[11], the following code shows how to write the address of the main_isr routine on the mtvec CSR and configure it to work in direct mode. Since the main_isr starts on an address that is a multiple of four, the two least significant bits of the address are zero, hence, by writing this value into the mtvec CSR we are configuring the mtvec.MODE subfield to work on the direct mode.

```
1 la   t0, main_isr  # Loads the main_isr routine address into t0
2 csrw mtvec, t0     # Copy t0 value into mtvec CSR
```

To configure the system to work with the vectored mode, the base address of the interrupt vector table may be loaded into a register and the least significant bit before writing the register's value into the mtvec CSR. The following code illustrates this process. In this case, the base address of the interrupt vector table, represented by the ivt label, is first loaded into register t0. Then, its least significant bit is set by the ori instruction and the final valued written into the mtvec CSR using the csrw instruction.

```
1 la   t0, ivt      # Load the interrupt vector table address into t0
2 ori  t0, t0, 0x1  # Set the least significant bit (MODE = vectored)
3 csrw mtvec, t0    # Copy t0 value into mtvec CSR
```

Setting-up the ISR stack

To set the ISR stack up, the system may allocate space on main memory and set the mscratch register so it points to the top of the ISR stack. The following code illustrates this process. First, the code allocates a 1024 byte array on the .bss section starting on an address that is a multiple of sixteen[12] . Then, the initialization code, indicated by the start label, loads the top of the ISR stack address into the t0 and copies its value into the mscratch CSR.

```
1  .section .bss
2  .align 4
3  isr_stack:
4  .skip 1024
5  isr_stack_end:
6
7  .section .text
8  .align 2
9  start:
10 la   t0, isr_stack_end
11 csrw mscratch, t0
```

Enabling interrupts

Once peripherals that generate interrupt signals are properly configured, and the interrupt service routine and the ISR stack are set, the initialization code must enable the mie.MEIE and the mstatus.MIE subfields to allow the CPU to handle external interrupts. The following code shows how this process can be performed.

```
1 # Enable external interrupts (mie.MEIE <= 1)
2 csrr t0, mie      # Read the mie register
```

[11]This is usually the case when programming for the RV32I ISA, since the architecture manual specifies that RV32I instructions must be stored on addresses that are multiples of four.

[12]The ilp32 ABI specifies that the stack pointer must always contain an address that is a multiple of 16.

```
3 li t2, 0x800      # Set the MEIE field (bit 11)
4 or t1, t1, t2
5 csrw mie, t1      # Update the mie register
6
7 # Enable global interrupts (mstatus.MIE <= 1)
8 csrr t0, mstatus  # Read the mstatus register
9 ori t0, t0, 0x8   # Set MIE field (bit 3)
10 csrw mstatus, t0  # Update the mstatus register
```

Chapter 11

Software Interrupts and Exceptions

As discussed in Section 5, many computer systems are organized so that the software is divided into user and system software. The system software (*e.g.*, the operating system kernel and device drivers) is the software responsible for protecting and managing the whole system, including interacting with peripherals to perform input and output operations and loading and scheduling user applications for execution. The user software is usually limited to performing operations with data that is located on registers and the main memory. Whenever the user software needs to perform a procedure that requires interacting with other parts of the system, such as reading data from a file or showing information on the computer display, it invokes the system software to perform the procedure on its behalf.

This chapter discusses the hardware mechanisms that protect the system from faulty or malicious user programs and how to program these mechanisms.

11.1 Privilege Levels

The Instruction Set Architecture defines the set of resources that the software can use to perform the intended computation. For example, it defines the set of instructions, their behavior, and operands, including the set of registers.

The **privilege level** defines which ISA resources (registers, instructions, *etc.*) are accessible by the software. They can be used to restrict the software execution and protect the system from software that attempts to perform operations not permitted. For example, the system can be configured to execute user applications with restricted privilege levels to prevent them from directly interacting with peripherals or accessing special microprocessor registers. The RISC-V Instruction Set Architecture defines three privilege levels:

- U: User/Application;
- S: Supervisor; and
- M: Machine

The **Machine** privilege level has the highest privileges, allowing full access to the hardware. The **Supervisor** privilege level has the second-highest privileges, and the **User/Application** privilege level has the least privileges.

A RISC-V hardware platform may implement a subset or all of these privilege levels. For example, when implementing a hardware platform for a compact and straightforward embedded system, only the Machine privilege level may be required. On the other hand, when implementing a hardware platform for a system that relies on an operating system to manage applications (*e.g.*, a computer desktop), it is usually useful to include all three privilege levels to facilitate the operating system implementation.

The RISC-V **privilege mode** defines the privilege level for the currently executing software. For example, when the Machine privilege mode is active, the currently executing software has Machine privilege levels and, hence, full access to the hardware.

The **unprivileged mode** is the privilege mode with the least privileges. In RISC-V, the unprivileged mode is the User/Application privilege mode, also known as the **user-mode** or **U-mode**. The **unprivileged ISA** is the sub-set of the Instruction Set Architecture accessible by the software running on unprivileged mode.

To simplify the discussion, the remaining of this chapter will focus on RISC-V processors that have only two privilege modes: User/Application and Machine mode.

11.2 Protecting the system

The User/Application mode limits the resources that can be accessed by the currently executing software; hence, to protect the system from faulty or malicious user programs, the system software usually sets the privilege mode to User/Application mode before executing (or returning control to) user code.

The following actions are usually taken to protect the system:

- **Configuring the system**: on power on, the hardware automatically sets the privilege mode to Machine mode and starts executing the boot code. The boot code loads the operating system software into memory and invokes its initialization code in Machine mode, which allows the operating system to configure the whole system.
- **Executing user code**: once the system is set, the operating system may load user programs into main memory and execute them. However, before transferring control to execute the user code, it sets the privilege mode as User/Application mode.
- **Handling illegal operations**: in case the user software tries to perform a privileged operation, such as interacting with peripherals, the hardware stops executing the user code and invokes the operating system so it can handle the illegal operation. The hardware transfer the control to the operating system using the exception handling mechanism, as discussed in Section 11.3.
- **Invoking the operating system**: if the user program needs to perform a sensitive procedure, such as an output to a peripheral, it must invoke the operating system, which will perform the procedure on the user-program behalf. When transferring control to the operating system, the hardware must change the privilege mode to Supervisor or Machine mode so the operating system may execute with proper privilege. To do so, ISAs usually include a mechanism, called software interrupt, that allows code running on unprivileged mode to invoke the system code and change the privilege mode at the same time. This mechanism is designed so that the user code may not change the privilege mode and execute its own code. Once the operating system finishes performing the procedure on the user-program behalf, it may change the privilege mode back to the User/Application mode and return to the user program.
- **Handling external interrupts**: upon an external interrupt, the hardware sets the privilege mode as Machine mode, so the interrupt service routine has enough privilege to handle the interrupt. Notice that the interrupt service routines belongs to the system software.

11.3 Exceptions

Exceptions are events generated by the CPU in response to exceptional conditions when executing instructions. Trying to execute an illegal instruction[1], for example, is a condition that causes a RISC-V CPU to generate an exception.

[1] An illegal instruction is an instruction that is not recognized by the CPU.

Exceptions usually trigger an exception handling mechanism so that the exceptional condition may be dealt with before the CPU may continue executing the program. This mechanism normally causes the CPU to redirect the execution flow to a system routine that may:

1. save the current program context;

2. handle the exceptional condition; and

3. restore the context of the saved program to continue the execution[2].

Notice that the exception handling flow is very similar to the one employed to handle hardware interrupts. In fact, RISC-V CPUs use the same mechanism to handle both interrupts and exceptions, *i.e.*, it saves part of the current context (*e.g.*, PC contents), sets the `mcause` CSR, and redirects the execution flow to an interrupt service routine. As discussed in Section 10.3.4, the interrupt service routine can distinguish between an interrupt and an exception by inspecting the `mcause` Control and Status Register. More specifically, the `mcause.INTERRUPT` CSR field indicates whether the CPU is handling an interrupt or an exception. Also, the `mcause.EXCCODE` CSR field indicates the source of the interrupt or the exception. There may be several sources of exceptions and interrupts on RISC-V. Table 11.1 shows the sources of interrupts and exceptions[3] and their respective codes on the `mcause` CSR.

The exception handling mechanism is usually employed to protect the system from illegal user code operations. In this context, the hardware is configured by the system software to generate exceptions in case the privilege mode is set as User/Application and the CPU tries to execute certain operations, such as accessing addresses that are mapped to peripheral devices or accessing Control and Status Registers that can only be accessed in Machine mode. Upon an exception, the interrupt service routine, which belongs to the system software, may decide what to do with the user program.

> **NOTE:** Exceptions occur as a result of executing an instruction; hence, they are synchronous events. Interrupts may occur at any time, independently of the CPU execution cycle; therefore, they are asynchronous events.

11.4 Software Interrupts

Software interrupts are events generated by the CPU when it executes special instructions. For example, in RISC-V, the environment call (`ecall`) and the breakpoint (`break`) instructions cause the CPU to generate software interrupts upon execution. They are similar to exceptions in the sense that they are synchronous events that occur due to executing an instruction. Nonetheless, exceptions are only generated on exceptional conditions, while software interrupts are always generated when the CPU executes these special instructions.

Software interrupts usually trigger a mechanism that changes the privilege mode and redirects the execution to a routine designed to handle the interrupt. This mechanism allows user programs to invoke a system software that requires a higher privilege level.

Most ISAs employ the same mechanism to handle interrupts, exceptions, and software interrupts, *i.e.*, they save part of the current context (*e.g.*, PC contents), sets the cause (*e.g.*, the `mcause` CSR), and redirect the execution flow to an interrupt service routine.

[2]Depending on the exception, the system may decide to terminate the program that was executing. In these cases, there is no need to restore the program context.

[3]Some of these exceptions and interrupts are only caused on system that implement the User/Application and the Supervisor privilege modes.

mcause fields		Cause
INTERRUPT	EXCCODE	
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	Reserved for future standard use
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	Reserved for future standard use
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	Reserved for future standard use
1	11	Machine external interrupt
1	12-15	Reserved for future standard use
1	$\geq$ 16	Reserved for platform use
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Reserved
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved for future standard use
0	15	Store/AMO page fault
0	16-23	Reserved for future standard use
0	24-31	Reserved for custom use
0	32-47	Reserved for future standard use
0	48-63	Reserved for custom use
0	$\geq$64	Reserved for future standard use

Table 11.1: Sources of interrupts and exceptions and their codes on the `mcause` CSR.

11.5 Protecting RISC-V systems

The following sections discuss how to use the RISC-V privilege modes and the exception and software interrupt handling mechanisms to protect the system from faulty or malicious user software.

11.5.1 Changing the privilege mode

RISC-V CPUs store the current privilege mode on an internal storage device that is not directly visible to software. In other words, the software may not directly inspect or modify this storage device to identify or manipulate the current privilege mode. The only way to inspect the current privilege mode is by generating a software interrupt or exception, which copies the privilege mode code into the `mstatus` Machine Previous Privilege field (`mstatus.MPP`). Also, the only way to set the current privilege mode is to modify the contents of the `mstatus` Machine Previous Privilege field (`mstatus.MPP`) and execute the `mret` instruction, which uses the value in the `mstatus.MPP` CSR field to set the current privilege mode.

The following code shows how the system software may change the privilege mode to User/Application mode and simultaneously invoke a user program. First, it sets the `mstatus.MPP` CSR field with "00", the User/Application mode code. Then, it loads the user software entry point into the `mepc` CSR. Finally, it executes the `mret` instruction, which changes the mode using the code in the `mstatus.MPP` CSR field and changes the program counter using the value in the `mepc` CSR at the same time.

```
1  # Changing to User/Application mode
2  csrr t1, mstatus     # Update the mstatus.MPP
3  li t2, ~0x1800       #  field (bits 11 and 12)
4  and t1, t1, t2       #  with value 00 (U-mode)
5  csrw mstatus, t1
6
7  la t0, user_main     # Loads the user software
8  csrw mepc, t0        # entry point into mepc
9
10 mret                 # PC <= MEPC; mode <= MPP;
```

11.5.2 Configuring the exception and software interrupt mechanisms

RISC-V CPUs use a similar mechanism to handle interrupts, exceptions, and software interrupts, *i.e.*, it saves part of the current context (*e.g.*, the PC contents), sets the `mcause` and other CSRs, and redirects the execution flow to an interrupt service routine. Consequently, configuring the exception and software interrupt handling mechanisms is very similar to configuring the external interrupt handling mechanism, as discussed in Section 10.3.5.

The system software configures the exception and software interrupt mechanisms by registering the routines that will handle these events. This is performed in the same way that interrupt service routines are registered to handle external interrupts. In the direct mode, a single routine is registered and this routine is responsible for inspecting the `mcause` CSR to identify the event source and invoke the proper routine. In the vectored mode, the external interrupt, exception, and software interrupt handling routines must be registered on the interrupt vector table. Section 10.3.5 shows fragments of code that configure the interrupt handling mechanism in direct and vectored modes.

On RISC-V, external interrupts must be enabled by setting the `mstatus` and the `mie` Control and Status Registers. On the other hand, exceptions and software interrupts are always enabled and do not need extra configuration.

11.5.3 Handling illegal operations

RISC-V CPUs generate exceptions whenever an instruction tries to execute an illegal operation, such as executing an instruction that the CPU does not recognize[4].

RISC-V systems that contain the Machine and the User/Application modes usually include a memory protection unit [5] that can be configured to generate exceptions whenever the CPU tries to read or write data or fetch instructions for execution from specific addresses. The operating system may protect the system by configuring this unit to generate exceptions whenever code executing in User/Application mode tries to access protected addresses, such as addresses mapped to peripherals and memory addresses that contain the operating system or other software. In these cases, if the CPU tries to read/write data from/to a protected address, the memory protection unit generates a "Load access fault"/"Store/AMO access fault" exception (`mstatus.EXCCODE`=5/7). Also, if the CPU tries to fetch an instruction for execution

[4]This operation generates an "Illegal instruction" exception, which is identified by value two on the `mcause.EXCCODE` CSR field.

from a protected address, the system generates an "Instruction access fault" exception (`mstatus.EXCCODE`=1).

Whenever an exception is generated, the RISC-V CPU:

- Saves the current program counter into the `mepc` CSR.
- Sets the `mcause` CSR with the code that identifies the exception source.
- Saves the current mode into the `mstatus.MPP` CSR field.
- Changes the mode to Machine mode.
- Sets the program counter to redirect the execution to the exception handling routine.

Some exceptions may also set the Machine Trap Value (`mtval`) CSR with extra information about the exception. For example, when a load or store access exception occurs, the `mtval` CSR is set with the faulting virtual address.

Depending on the exception, it may make sense to fix the problem that caused the exception and return the execution to the software that caused the exception to continue its execution. Page faults are examples of exceptions that can be handled by the system, so the software that caused the exception may continue its execution. In these cases, handling the exception is usually similar to handling an external interrupt, *i.e.* the exception handling routine must save the context, handle the exception, and, finally, recover the context so the software running on the CPU may continue its execution. In cases where illegal operations generate exceptions, and there is no known way to recover from the problem, the operating system may kill the offending process, *i.e.*, the process that tried to execute the illegal operation.

11.5.4 Handling system calls

As discussed in Section 6.7.4, in RISC-V, the user code may invoke the operating system, *i.e.*, perform a system call, by executing the environment call (`ecall`) instruction. This instruction generates a software interrupt, which invokes the interrupt and exception handling mechanism. In case an `ecall` instruction is executed on the User/Application mode, the hardware sets the `mcause.INTERRUPT` CSR field with zero and the `mcause.EXCCODE` CSR field with eight. Hence, if the interrupt/exception handling mechanism is configured in direct mode, the main interrupt service routine may identify a call to the operating system by comparing the value at the `mcause` register with eight.

Whenever an exception or a software interrupt occurs, the system records the program counter contents on `mepc` CSR. After handling an exception, for example, a page fault, the system may return the execution to the same instruction that caused the exception, so it may try to perform its operation again. Nonetheless, on software interrupts, the system must not return the execution to the same instruction; otherwise, it would invoke the operating system again. In this case, the system must return to the subsequent instruction. To do so, the software interrupt handling routine must adjust the value in `mepc` to point to the next instruction before executing the `mret` instruction.

The following code shows how to adjust the `mepc` CSR to point to the next instruction before executing the `mret` instruction.

```
1  # Adjusting MEPC so mret returns to the instruction
2  # placed after the ecall instruction that
3  # generated the software interrupt
4  csrr a1, mepc  # load mepc into a1
5  addi a1, a1, 4 # adds 4 to a1
6  csrw mepc, a1  # writes the new address to mepc
```

Bibliography

[1] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture.*, September 2016.

[2] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2A: Instruction Set Reference, A-L.*, September 2016.

[3] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2B: Instruction Set Reference, M-U.*, September 2016.

[4] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual. Volume i: Unprivileged isa, version 20191213. Technical report, SiFive Inc., 2019.

[5] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual. Volume II: Privileged architecture, document version 20190608-priv-msu-ratified. Technical report, SiFive Inc., 2019.

Appendix A

RV32IM ISA reference card

The next pages contain a reference card for the RV32IM ISA.

RV32IM assembly instructions reference card

Prof. Edson Borin
Institute of Computing - Unicamp

RV32IM registers (prefix x) and their aliases

x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15
zero	ra	sp	gp	tp	t0	t1	t2	s0	s1	a0	a1	a2	a3	a4	a5
x16	x17	x18	x19	x20	x21	x22	x23	x24	x25	x26	x27	x28	x29	x30	x31
a6	a7	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	t3	t4	t5	t6

Main control status registers

CSRs:	mtvec	mepc	mcause	mtval	mstatus	mscratch
Fields of mstatus:	mie	mpie	mip			

Logic, Shift, and Arithmetic instructions	
`and rd, rs1, rs2`	Performs the bitwise "and" operation on `rs1` and `rs2` and stores the result on `rd`.
`or rd, rs1, rs2`	Performs the bitwise "or" operation on `rs1` and `rs2` and stores the result on `rd`.
`xor rd, rs1, rs2`	Performs the bitwise "xor" operation on `rs1` and `rs2` and stores the result on `rd`.
`andi rd, rs1, imm`	Performs the bitwise "and" operation on `rs1` and `imm` and stores the result on `rd`.
`ori rd, rs1, imm`	Performs the bitwise "or" operation on `rs1` and `imm` and stores the result on `rd`.
`xori rd, rs1, imm`	Performs the bitwise "xor" operation on `rs1` and `imm` and stores the result on `rd`.
`sll rd, rs1, rs2`	Performs a logical left shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the value on `rs2`.
`srl rd, rs1, rs2`	Performs a logical right shift on the value at `rs1` and stores the result on `rd`. The amount of right shifts is indicated by the value on `rs2`.
`sra rd, rs1, rs2`	Performs an arithmetic right shift on the value at `rs1` and stores the result on `rd`. The amount of right shifts is indicated by the value on `rs2`.
`slli rd, rs1, imm`	Performs a logical left shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.
`srli rd, rs1, imm`	Performs a logical right shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.
`srai rd, rs1, imm`	Performs an arithmetic right shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.
`add rd, rs1, rs2`	Adds the values in `rs1` and `rs2` and stores the result on `rd`.
`sub rd, rs1, rs2`	Subtracts the value in `rs2` from the value in `rs1` and stores the result on `rd`.
`addi rd, rs1, imm`	Adds the value in `rs1` to the immediate value `imm` and stores the result on `rd`.
`mul rd, rs1, rs2`	Multiplies the values in `rs1` and `rs2` and stores the result on `rd`.
`div{u} rd, rs1, rs2`	Divides the value in `rs1` by the value in `rs2` and stores the result on `rd`. The `U` suffix is optional and must be used to indicate that the values in `rs1` and `rs2` are unsigned.
`rem{u} rd, rs1, rs2`	Calculates the remainder of the division of the value in `rs1` by the value in `rs2` and stores the result on `rd`. The `U` suffix is optional and must be used to indicate that the values in `rs1` and `rs2` are unsigned.

Unconditional control-flow instructions	
`j lab`	Jumps to address indicated by symbol `sym` (Pseudo-instruction).
`jr rs1`	Jumps to the address stored on register `rs1` (Pseudo-instruction).
`jal lab`	Stores the return address (PC+4) on the return register (`ra`), then jumps to label `lab` (Pseudo-instruction).
`jal rd, lab`	Stores the return address (PC+4) on register `rd`, then jumps to label `lab`.
`jarl rd, rs1, imm`	Stores the return address (PC+4) on register `rd`, then jumps to the address calculated by adding the immediate value `imm` to the value on register `rs1`.
`ret`	Jumps to the address stored on the return register (`ra`) (Pseudo-instruction).
`ecall`	Generates a software interruption. Used to perform system calls.
`mret`	Returns from an interrupt handler.

Conditional set and control-flow instructions	
`slt rd, rs1, rs2`	Sets `rd` with 1 if the signed value in `rs1` is less than the signed value in `rs2`, otherwise, sets it with 0.
`slti rd, rs1, imm`	Sets `rd` with 1 if the signed value in `rs1` is less than the sign-extended immediate value `imm`, otherwise, sets it with 0.
`sltu rd, rs1, rs2`	Sets `rd` with 1 if the unsigned value in `rs1` is less than the unsigned value in `rs2`, otherwise, sets it with 0.
`sltui rd, rs1, imm`	Sets `rd` with 1 if the unsigned value in `rs1` is less than the unsigned immediate value `imm`, otherwise, sets it with 0.
`seqz rd, rs1`	Sets `rd` with 1 if the value in `rs1` is equal to zero, otherwise, sets it with 0 (Pseudo-instruction).
`snez rd, rs1`	Sets `rd` with 1 if the value in `rs1` is not equal to zero, otherwise, sets it with 0 (Pseudo-instruction).
`sltz rd, rs1`	Sets `rd` with 1 if the signed value in `rs1` is less than zero, otherwise, sets it with 0 (Pseudo-instruction).
`sgtz rd, rs1`	Sets `rd` with 1 if the signed value in `rs1` is greater than zero, otherwise, sets it with 0 (Pseudo-instruction).
`beq rs1, rs2, lab`	Jumps to label `lab` if the value in `rs1` is equal to the value in `rs2`.
`bne rs1, rs2, lab`	Jumps to label `lab` if the value in `rs1` is different from the value in `rs2`.
`beqz rs1, lab`	Jumps to label `lab` if the value in `rs1` is equal to zero (Pseudo-instruction).
`bnez rs1, lab`	Jumps to label `lab` if the value in `rs1` is not equal to zero (Pseudo-instruction).
`blt rs1, rs2, lab`	Jumps to label `lab` if the signed value in `rs1` is smaller than the signed value in `rs2`.
`bltu rs1, rs2, lab`	Jumps to label `lab` if the unsigned value in `rs1` is smaller than the unsigned value in `rs2`.
`bge rs1, rs2, lab`	Jumps to label `lab` if the signed value in `rs1` is greater or equal to the signed value in `rs2`.
`bgeu rs1, rs2, lab`	Jumps to label `lab` if the unsigned value in `rs1` is greater or equal to the unsigned value in `rs2`.

Data movement instructions	
`mv rd, rs`	Copies the value from register `rs` into register `rd` (Pseudo-instruction).
`li rd, imm`	Loads the immediate value `imm` into register `rd` (Pseudo-instruction).
`la rd, rot`	Loads the label address `rot` into register `rd` (Pseudo-instruction).
`lw rd, imm(rs1)`	Loads a 32-bit `signed` or `unsigned word` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lh rd, imm(rs1)`	Loads a 16-bit `signed halfword` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lhu rd, imm(rs1)`	Loads a 16-bit `unsigned halfword` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lb rd, imm(rs1)`	Loads a 8-bit `signed byte` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lbu rd, imm(rs1)`	Loads a 8-bit `unsigned byte` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`sw rs1, imm(rs2)`	Stores the 32-bit value at register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.
`sh rs1, imm(rs2)`	Stores the 16 least significant bits from register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.
`sb rs1, imm(rs2)`	Stores the 8 least significant bits from register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.
`L{W|H|HU|B|BU} rd, lab`	For each one of the `lw`, `lh`, `lhu`, `lb`, and `lbu` machine instructions there is a pseudo-instruction that performs the same operation, but the memory address is calculated based on a label (`lab`) (Pseudo-instruction).
`S{W|H|B} rd, lab`	For each one of the `sw`, `sh`, and `sb` machine instructions there is a pseudo-instruction that performs the same operation, but the memory address is calculated based on a label (`lab`) (Pseudo-instruction).

Control and Status Read and Write instructions	
`csrr rd, csr`	Copies the value from the control and status register `csr` into register `rd` (Pseudo-instruction).
`csrw csr, rs`	Copies the value from register `rs` into the control and status register `csr` (Pseudo-instruction).
`csrrw rd, csr, rs1`	Copies the value from the control and status register `csr` into register `rd` and the value from the `rs1` register to the control and status register `csr`. If `rd`=`rs1`, the instruction performs an atomic swap between registers `csr` and `rs1`..
`csrc csr, rs`	Clears control and status register (`csr`) bits using the contents of the `rs` register as a bit mask. (Pseudo-instruction).
`csrs csr, rs`	Sets control and status register (`csr`) bits using the contents of the `rs` register as a bit mask. (Pseudo-instruction).