3
Assembly Language

Assembly language can broadly be defined as a textual representation of machine instructions. Before building a processor, we need to know about the semantics of different machine instructions, and a rigorous study of assembly language will be of benefit in this regard. An assembly language is specific to an ISA and compiler framework; hence, there are many flavors of assembly languages. In this chapter we shall describe the broad principles underlying different variants of assembly languages, some generic concepts and terms. We will subsequently design our own assembly language, *SimpleRisc* . It is a simple RISC ISA with a few instructions. Subsequently, in Chapter 8, we will design a processor that fully implements this ISA. Thus, the plan for this chapter is as follows. We shall first convince ourselves of the need for assembly language in Section 3.1 from the point of view of both software developers and hardware designers. Then we shall proceed to discuss the generic semantics of assembly languages in Section 3.2. Once, we have a basic understanding of assembly languages, we shall design our own assembly language, *SimpleRisc* , in Section 3.3, and then design a method to encode it using a sequence of 32 bits in Section 3.3.14.

Subsequently, in Chapter 4 we shall describe the ARM assembly language that is meant for ARM based processors, and in Chapter 5, we shall describe the x86 assembly language meant for Intel/AMD processors. In these two chapters, these machine specific assembly languages will be covered in great detail. This chapter is introductory, and creates the framework for a more serious study of different instruction sets and assembly languages.

3.1 Why Assembly Language

3.1.1 Software Developer's Perspective

A human being understands natural languages such as English, Russian, and Spanish. With some additional training a human can also understand computer programming languages such as C or Java. However, a computer is a dumb machine as mentioned in Chapter 1. It is

not smart enough to understand commands in a human language such as English, or even a programming language such as C. It only understands zeros and ones. Hence, to program a computer it is necessary to give it a sequence of zeros and ones. Indeed some of the early programmers used to program computers by turning on or off a set of switches. Turning on a switch corresponded to a 1, and turning it off meant a 0. However, for today's massive multi-million line programs, this is not a feasible solution. We need a better method.

Consequently, we need an automatic converter that can convert programs written in high level languages such as C or Java to a sequence of zeros and ones known as *machine code*. Machine code contains a set of instructions known as *machine instructions*. Each machine instruction is a sequence of zeros and ones, and instructs the processor to perform a certain action. A program that can convert a program written in a high level language to machine code is called a *compiler*(see Figure 3.1).

Definition 24

- *A high level programming language such as C or Java uses fairly complex constructs and statements. Each statement in these languages typically corresponds to a multitude of basic machine instructions. These languages are typically independent of the processor's ISA.*
- *A compiler is an executable program that converts a program written in a high level language to a sequence of machine instructions that are encoded using a sequence of zeros and ones.*

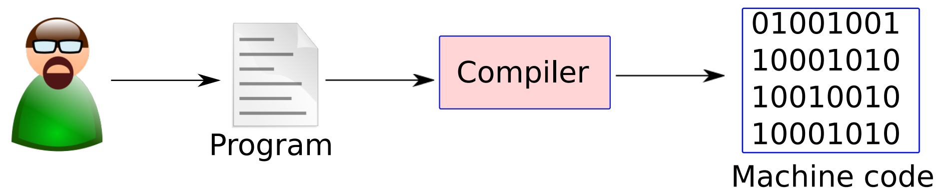


Figure 3.1: The compilation process

Note that the compiler is an executable program that typically runs on the machine that it is supposed to generate machine code for. A natural question that can arise is – who wrote the first compiler? See Trivia 1.

Trivia 1 *Who Wrote the First Compiler? If a programmer wrote the compiler in a high level language such as C or Java, then she must have needed a compiler to compile it into machine code. However, she did not have a compiler with her at that point of time, because*

she was in the process of building one! Since she did not have a compiler, while building the compiler, how did she ultimately build it? This is an example of a chicken and egg problem. *The classic chicken and egg problem poses a simple yet vexing question – did the chicken come first or the egg come first? However, the chicken and egg problem has a solution that can be explained in terms of evolution. Scientists believe that early organisms reproduced by replication. At some point of time, due to a genetic mutation, an organism started to lay eggs. These organisms perpetuated, and started reproducing by only laying eggs. They evolved into all kinds of birds and reptiles, including chickens.*

We can explain this conundrum in a similar manner. The early programmers wrote simple compilers using machine instructions. A primitive compiler is just a sequence of zeros and ones. The early programmers then used these primitive compilers to compile programs. A special class of such programs were compilers themselves. They were written in high level languages and were better in terms of features, functionality and even performance. These first generation compilers were then used to create second generation compilers, and this process has continued till date. Nowadays, if a new processor is being developed, then it is not necessary to follow this procedure. Programmers, use another set of programs called cross compilers. *A cross compiler runs on an existing processor, and produces an executable using the machine instructions of the new processor that is being developed. Once the new processor is ready, this program can be moved to the new processor and executed directly. It is thus possible to develop a large range of software including compilers for processors with new instruction sets. Hence, most modern day programmers do not have to write programs using raw machine instructions.*

Definition 25
A cross compiler *is a program that runs on machine A, and generates machine code for machine B. It is possible that B has a different ISA.*

Given the ubiquity of compilers, almost all programs are written in high level languages and compilers are used to convert them to machine code. However, there are important exceptions to this rule. Note that the role of a compiler is two fold. First, it needs to correctly translate a program in a high level language to machine instructions. Second, it needs to produce efficient machine code that does not take a lot of space, and is fast. Consequently, algorithms in compilers have become increasingly complicated over the years. However, it is not always possible to meet these requirements. For example, in some scenarios, compilers might not be able to produce code that is fast enough, or has a certain kind of functionality that the programmer desires. Let us elaborate further. Algorithms in compilers are limited by the amount of analysis that they can perform on the program. For example, we do not want the process of compilation to be extremely slow. A lot of the problems in the area of compilers are computationally difficult to solve and are thus time consuming. Secondly, the compiler is not aware of the broad patterns in the code. For example, it is possible that a certain variable might only take a restricted set

of values, and on the basis of this, it might be possible to optimise the machine code further. It is hard for a compiler to figure this out. However, smart programmers can sometimes produce machine code that is more optimal than a compiler because they are aware of some broad patterns of execution, and their brilliant brains can outsmart compilers.

Secondly, it is also possible that a processor vendor might add new instructions in their ISA. In this case, compilers meant for older versions of the processor might not be able to leverage the new instructions. It will be necessary to add them manually in programs. Continuing this argument further, we observe that popular compilers such as gcc (GNU compiler collection) are fairly generic. They do not use all possible machine instructions that a processor provides while generating machine code. Typically, a lot of the missed out instructions are required by operating systems and device drivers (programs that interface with devices such as the printer, and scanner). These software programs require these instructions because they need low level access to the hardware. Consequently, system programmers have a strong incentive to occasionally bypass the compiler.

In all of these situations, it is necessary for programmers to manually embed a sequence of machine instructions in a program. As mentioned, there are two primary reasons for doing so – efficiency and extra functionality. Hence, from the point of view of system software developers, it is necessary to know about machine instructions such that they can be more productive in their job.

Now, our aim is to insulate modern day programmers from the intricate details of zeros and ones. Ideally, we do not want our programmers to program by manually turning on and off switches as was done fifty years ago. Consequently, a low level language called *assembly language* was developed (see Definition 26). Assembly language is a human readable form of machine code. Each assembly language statement typically corresponds to one machine instruction. Furthermore, it eases the burden on the programmer significantly by not forcing her to remember the exact sequence of zeros/ones that are needed to encode an instruction.

Definition 26

- *A low level programming language uses simple statements that correspond to typically just one machine instruction. These languages are specific to the ISA.*
- *The term "assembly language" refers to a family of low level programming languages that are specific to each ISA. They have a generic structure that consists of a sequence of assembly statements. Typically, each assembly statement has two parts – (1) an instruction code that is a mnemonic for a basic machine instruction, and (2) and a list of operands.*

From a practical standpoint, it is possible to write stand alone assembly programs and convert them to executables using a program called an *assembler*(Definition 27). Alternatively, it is also possible to embed snippets of assembly code in high level languages such as C or C++. The latter is more common. A compiler ensures that it is able to compile the combined program into machine code. The benefits of assembly languages are manifold. Since each

line in assembly code corresponds to one machine instruction, it is as expressive as machine code. Because of this one to one mapping, we do not sacrifice efficiency by writing programs in assembly. Secondly, it is a human readable and elegant form of textually representing machine code. It makes it significantly easier to write programs using it, and it is also possible to cleanly embed snippets of assembly code in software written in high level languages such as C. The third advantage of assembly language is that it defines a level of abstraction over and above real machine code. It is possible that two processors might be compatible with the same variant of assembly language, but actually have different machine encodings for the same instruction. In this case, assembly programs will be compatible across both of these processors.

Definition 27
An assembler is an executable program that converts an assembly program into machine code.

Example 20
The core engines of high performance 3D games need to be optimised for speed as much as possible [Phelps and Parks, 2004]. Most compilers fail to produce code that runs fast enough. It becomes necessary for programmers to manually write sequences of machine instructions.

Example 21
Vranas et. al. [Vranas et al., 2006] describe a high performance computing application to study the structure of an atomic nucleus. Since the computational requirements are high, they needed to run their program on a supercomputer. They observed that the core of the program lies in a small set of functions that are just 1000 lines long. They further observed that compilers were not doing a good in job in optimising the output machine code. Consequently, they decided to write the important functions in assembly code, and obtained record speedups on a supercomputer. Durr et. al. [Durr et al., 2009] subsequently used this framework to accurately calculate the mass of a proton and a neutron from first principles. The results were in complete agreement with experimentally observed values.

3.1.2 Hardware Designer's Perspective

The role of hardware designers is to design processors that can implement all the instructions in the ISA. Their main aim is to design an efficient processor that is optimal with regards to area, power efficiency, and design complexity. From their perspective, the ISA is the crucial link between software and hardware. It answers the basic question for them – "what to build?"

Hence, it is very essential for them to understand the precise semantics of different instruction sets such that they can design processors for them. As mentioned in Section 3.1.1, it is cumbersome to look at instructions as merely a sequence of zeros and ones. They can gain a lot by taking a look at the textual representation of a machine instruction, which is an assembly instruction.

An assembly language is specific to an instruction set and an assembler. In this chapter, we use the assembly language format of the popular GNU assembler [Elsner and Fenlason, 1994] to explain the syntax of a typical assembly language file. Note that other systems have similar formats, and the concepts are broadly the same.

3.2 The Basics of Assembly Language

3.2.1 Machine Model

Let us reconsider the basic abstract machine model explained in Chapter 1. We had finished the chapter by describing a form of the Harvard and Von Neumann machines with registers. Assembly languages do not see the instruction memory and data memory as different entities. They assume an abstract Von Neumann machine augmented with registers.

Refer to Figure 3.2 for a pictorial representation of the machine model. The program is stored in a part of the main memory. The central processing unit (CPU) reads out the program instruction by instruction, and executes the instructions appropriately. The program counter keeps track of the memory address of the instruction that a CPU is executing. We typically refer to the program counter using the acronym – PC. Most instructions are expected to get their input operands from registers. Recall that every CPU has a fixed number of registers (typically < 64). However, a large number of instructions, can also get their operands from the memory directly. It is the job of the CPU to co-ordinate the transfers to and from the main memory and registers. Secondly, the CPU also needs to perform all the arithmetic/logical calculations, and liaise with external input/output devices.

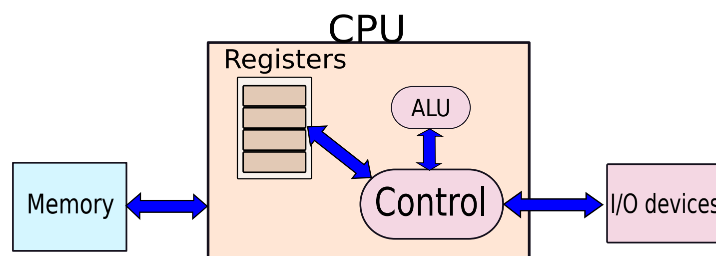


Figure 3.2: The Von Neumann machine with registers

Most flavors of assembly language assume this abstract machine model for a majority of their statements. However, since another aim of using assembly language is to have more fine grained and intrusive control of hardware, there are a fair number of assembly instructions that are cognisant of the internals of the processor. These instructions typically modify the behaviour of the processor by changing the behaviour of some key internal algorithms; they modify built-in

parameters such as power management settings, or read/write some internal data. Finally, note that the assembly language does not distinguish between machine independent and machine dependent instructions.

View of Registers

Every machine has a set of registers that are visible to the assembly programmer. ARM has 16 registers, x86 (32-bit) has 8 registers, and x86_64 (64 bits) has 16 registers. The registers have names. ARM names them $r0 \ldots r15$, and x86 names them $eax, ebx, ecx, edx, esi, edi, ebp$, and esp. A register can be accessed using its name.

In most ISAs, a return address register is used for function calls. Let us assume that a program starts executing a function. It needs to remember the memory address that it needs to come back to after executing the function. This address is known as the *return address*. Before jumping to the starting address of a function, we can save the value of the return address in this register. The return statement can simply be implemented by copying the value saved in the return address register to the PC. The return address register is visible to the programmer in assembly languages such as ARM and MIPS. However, x86 does not use a return address register. It uses another mechanism called a stack, which we shall study in Section 3.3.10.

In an ARM processor, the PC is visible to the programmer and it is the last register ($r15$). It is possible to read the value of the PC, as well as set its value. Setting the value of the PC means that we want to branch to a new location within the program. However, in x86, the program counter is implicit, and is not visible to the programmer.

3.2.2 View of Memory

In Section 1.6.7, we explained the concept of a memory in an abstract machine. The memory can be thought of as one large array of bytes. Each byte has a unique address, which is essentially its location in the array. The address of the first byte is 0, the address of the second byte is 1, and so on. Note that the finest granularity at which we can access memory is at the level of a byte. We do not have a method to uniquely address a given bit. The address is a 32-bit unsigned integer in 32-bit machines and it is a 64-bit unsigned integer in 64-bit machines.

Now, in a Von Neumann machine, we assume that the program is stored in memory as a sequence of bytes, and the program counter points to the next instruction that is going to be executed.

Assuming that memory is one large array of bytes is fine, if all our data items are only one byte long. However, languages such as C and Java have data types of different sizes – char (1 byte), short (2 bytes), integer (4 bytes), and long integer (8 bytes). For a multi-byte data type it is necessary to find a representation for it in memory. There are two possible ways of representing a multibyte data type in memory – *little endian* and *big endian*. Secondly, we also need to find methods to represent arrays or lists of data in memory.

Little Endian and Big Endian Representations

Let us consider the problem of storing an integer in locations 0-3. Let the integer be 0x87654321. It can be broken into four bytes – 87, 65, 43, and 21. One option is to store the most significant byte, 87, in the lowest memory address 0. The next location can store 65, then 43, and then

21. This is called the *big endian* representation because we are starting from the position of the most significant byte. In comparison, we can save the least significant byte first in location 0, and then continue to save the most significant byte in location 3. This representation is called *little endian*. Figure 3.3 shows the difference.

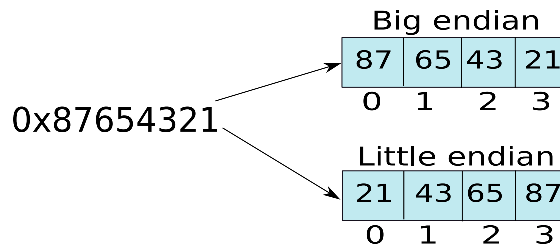


Figure 3.3: Big endian and little endian representations

There is as such no reason to prefer one representation over the other. It depends on the convention. For example, x86 processors use the little endian format. Early versions of ARM processors used to be little endian. However, now they are bi-endian. This means an ARM processor can work as both a little endian and a big endian machine depending on the settings set by the user. Traditionally, IBM® POWER® processors, and Sun® SPARC® processors have been big endian.

Representing Arrays

An array is a linearly ordered set of objects, where an object can be a simple data type such as an integer or character, or can be a complex data type also.

```
int a[100];
char c[100];
```

Let us consider a simple array of integers, a. If the array has 100 entries, then the total size of the array in memory is equal to $100 \times 4 = 400$ bytes. If the starting memory location of the array is loc. Then $a[0]$ is stored in the locations $(loc + 0), (loc + 1), (loc + 2), (loc + 3)$. Note that there are two methods of saving the data – big endian and little endian. The next array entry, $a[1]$, is saved in the locations $(loc + 4) \ldots (loc + 7)$. By continuing the argument further, we note that the entry $a[i]$ is saved in the locations – $(loc + 4 \times i) \ldots (loc + 4 \times i + 3)$.

Most programming languages define multidimensional arrays of the form:

```
int a[100][100];
char c[100][100];
```

They are typically represented as regular one dimensional arrays in memory. There is a mapping function between the location in a multidimensional array and an equivalent 1-dimensional array. Let us consider Example 22. We can extend the scheme to consider multidimensional arrays of dimensions greater than 2.

Example 22
Consider a multidimensional array: $a[100][100]$. Map each entry (i, j) to an entry in a 1-D array: $b[10000]$.

Answer: *Let us assume that each entry (i, j), is in a (row,column) format. Let us try to save the array in row-major fashion. We save the first row in contiguous locations, then the second row and so on. The starting entry of each row is equal to $100 \times i$. Within each row the offset for column j is equal to j. Thus we can map (i, j) to the entry: $(100 \times i + j)$ in the array b.*

We observe that a two-dimensional array can be saved as a one dimensional array by saving it in row-major fashion. This means that data is saved row wise. We save the first row, then the second row, and so on. Likewise, it is also possible to save a multidimensional array in column major fashion, where the first column is saved, then the second column and so on.

Definition 28

row major *In this representation, an array is saved row wise in memory.*

column major *In this representation, an array is saved column wise in memory.*

3.2.3 Assembly Language Syntax

In this section, we shall describe the syntax of assembly language. The exact syntax of an assembly file is dependent on the assembler. Different assemblers can use different syntax, even though they might agree on the basic instructions, and their operand formats. In this chapter, we explain the syntax of the GNU family of assembly languages. They are designed for the GNU assembler, which is a part of the GNU compiler collection (gcc). Like all GNU software, this assembler and the associated compiler is freely available for most platforms. As of 2012, the assembler is available at [gnu.org,]. In this section, we shall provide a brief overview of the format of assembly files. For additional details refer to the official manual of the GNU assembler [Elsner and Fenlason, 1994]. Note that other assemblers such as NASM, and MASM, have their own formats. However, the overall structure is not conceptually very different from what we shall describe in this section.

Assembly Language File Structure

An assembly file is a regular text file, and it has a (.s) suffix. The reader can quickly generate an assembly file for a C program (test.c), if she has the gcc (GNU Compiler) installed. It can be generated by issuing the following command.

```
gcc -S test.c
```

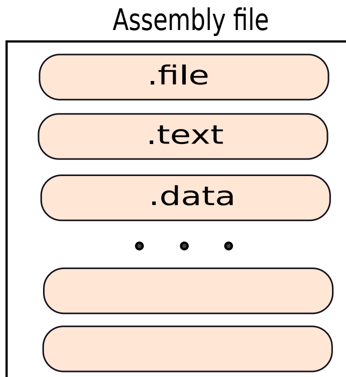


Figure 3.4: Assembly language file structure

The generated assembly file will be named test.s. GNU assembly files have a very simple structure, as shown in Figure 3.4. They contain a list of sections. Examples of different sections are text (actual program), data (data with initialised values), and bss (common data that is initialised to 0). Each section starts with a section heading, which is the name of the section prefixed by the '.' symbol. For example, the text section starts with the line ".text". Thereafter, there is a list of assembly language statements. Each statement is typically terminated by the newline character. Likewise, the data section contains a list of data values. An assembly file begins with the file section that contains a line of the form – ".file <name of the file> ". When we are generating an assembly file from a C program using the gcc compiler, the name of the file in the .file section is typically the same as our original C program (test.c). The text section is mandatory, and the rest of the sections are optional. There might be one or more data sections. It is also possible to define new sections using the .section directive. In this book, we primarily concentrate on the text section because we are interested in learning about the nature of the instruction set. Let us now look at the format of assembly statements.

Basic Statements

A bare bones assembly language statement specifies an assembly instruction and has two parts – the instruction and its list of operands, as shown in Figure 3.5. The instruction is a textual identifier of the actual machine instruction. The list of operands contains the value or location of each operand. The value of an operand is a numeric constant. It is also known as an *immediate* value. The operand locations can either be register locations or memory locations.

Figure 3.5: Assembly language statement

Definition 29
In computer architecture, a constant value specified in an instruction is also known as an immediate*.*

Now, let us consider an example.

```
add r3, r1, r2
```

In this ARM assembly statement, the *add* instruction is specifying the fact that we wish to add two numbers and save the result in some pre-specified location. The format of the add instruction in this case is as follows: $< instruction > < destination\, register > < operand\, register\, 1 > < operand\, register\, 2 >$. The name of the instruction is *add*, the destination register is $r3$, the operand registers are $r1$ and $r2$. The detailed steps of the instruction are as follows:

1. Read the value of register $r1$. Let us refer to the value as v_1.
2. Read the value of register $r2$. Let us refer to the value as v_2.
3. Compute $v_3 = v_1 + v_2$.
4. Save v_3 in register $r3$

Let us now give an example of two more instructions that work in a similar fashion(see Example 23).

Example 23
sub r3, r1, r2
mul r3, r1, 3

The *sub* instructions subtracts two numbers stored in registers, and the *mul* instruction multiplies a number stored in the register, $r1$, with the numeric constant, 3. Both the instructions save the result in the register, $r3$. Their mode of operation is similar to the *add* instruction. Moreover, the arithmetic instructions – *add*, *sub*, and *mul* – are also known as data processing instructions. There are several other classes of instructions such as data transfer instructions that load or store values from memory, and control instructions that implement branching.

Generic Statement Structure

The generic structure of an assembly statement is shown in Figure 3.6. It consists of three fields namely a label (identifier of the instruction), the key (an assembly instruction, or a directive to

the assembler), and a comment. All three of these fields are optional. However, any assembly statement needs to have at least one of these fields.

A statement can optionally begin with a label. A *label* is a textual identifier for the statement. In other words, a label uniquely identifies an assembly statement in an assembly file. Note that we are not allowed to repeat labels in the same assembly file. We shall find labels to be very useful while implementing branch instructions.

Definition 30
A label in an assembly file uniquely identifies a given point or data item in the assembly program.

An example of a label is shown in Example 24. Here the name of the label is "label1", and it is succeeded by a colon. After the label we have written an assembly instruction and given it a list of operands. A label can consist of valid alpha-numeric characters $[a-z][A-Z][0-9]$ and the symbols '.', '_', and '$'. Typically, we cannot start a label with a digit. After specifying a label we can keep the line empty, or we can specify a key (part of an assembly statement). If the key begins with a '.', then it is an assembler directive, which is valid for all computers. It directs the assembler to perform a certain action. This action can include starting a new section, or declaring a constant. The directive can also take a list of arguments. If the key begins with a letter, then it is a regular assembly instruction.

Example 24
label1: add r1, r2, r3

After the label, assembly instruction, and list of operands, it is possible to optionally insert comments. The GNU assembler supports two types of comments. We can insert regular C or Java style comments enclosed between /* and */. It is also possible to have a small single line comment by preceding the comment with the '@' character in ARM assembly.

Example 25

```
label1: add r1, r2, r3 @ Add the values in r2 and r3
label2: add r3, r4, r5 @ Add the values in r4 and r5
add r5, r6, r7         /* Add the values in r6 and r7 */
```

Let us not slightly amend our statement regarding labels. It is possible that an assembly statement only contains a label, and does not contain a key. In this case, the label essentially points to an empty statement, which is not very useful. Hence, the assembler assumes that in such a case a label points to the nearest succeeding assembly statement that contains a key.

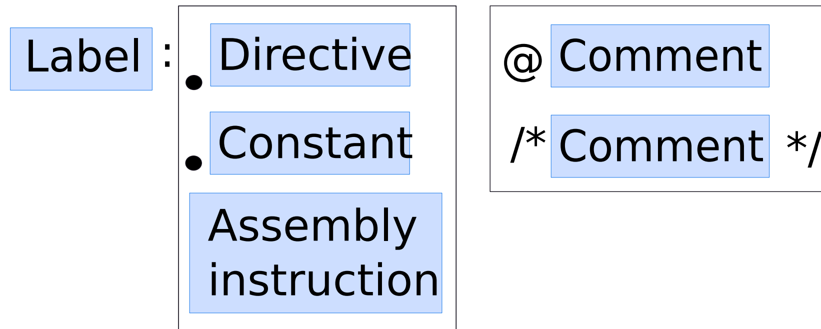


Figure 3.6: Generic Structure of an assembly statement

3.2.4 Types of Instructions

Classification by Functionality

The four major types of instructions are as follows:

1. **Data Processing Instructions:** Data processing instructions are typically arithmetic instructions such as add, subtract, and multiply, or logical instructions that compute bitwise or, and exclusive or. Comparison instructions also belong to this family.

2. **Data Transfer Instructions:** These instructions transfer values between two locations. A location can be a register or a memory address.

3. **Branch Instructions:** Branch instructions help the processor's control unit to jump to different parts of the program based on the values of operands. They are useful in implementing *for* loops and *if-then-else* statements.

4. **Exception Generating Instructions:** These specialised instructions help transfer control from a user level program to the operating system.

In this book we shall cover data processing, data transfer, and control instructions.

Classification based on the Number of Operands

As mentioned in Section 3.2.3, all assembly language statements in the GNU assembler have the same structure. They start with the name of the instruction, and are succeeded by a list of operands. We can classify instructions based on the number of operands that they require. If an instruction requires n operands, then we typically say that it is in the n-address format. For example, an instruction that does not require any operands is a 0-address format instruction. If it requires 3 operands, then it is a 3-address format instruction.

Definition 31
If an instruction requires n operands (including source and destination), then we say that it is a n-address format instruction.

In ARM most of the data processing instructions are in the 3-address format, and data transfer instructions are in the 2-address format. However, in x86 most of the instructions are in the 2-address format. The first question that comes to our mind is what is the logic of having a 3-address format instruction versus having a 2-address format instruction? There must be some tradeoff here.

Let us outline some general rules of thumb. If an instruction has more operands then it will require more bits to represent the instruction. Consequently, we will require more resources to store, and handle instructions. However, there is a flip side to this argument. Having more operands will also make the instruction more generic and flexible. It will make the life of compiler writers and assembly programmers much easier, because it will be possible to do more things with an instruction that uses more operands. The reverse logic applies to instructions that take less operands. They take less space to store, and are less flexible. Let us consider an example. Assume that we are trying to add two numbers, 3 and 5, to produce a result, 8.

An ARM instruction for addition would look like this:

```
add r3, r1, r2
```

This instruction adds the contents of registers, $r1(3)$, and $r2(5)$, and saves it in $r3(8)$. However, an x86 instruction would look like this:

```
add edx, eax
```

Here, we assume that *edx* contains 3, and *eax* contains 5. The addition is performed, and the result, 8, is stored back in *edx*. Thus, in this case the x86 instruction is in the 2-address format because the destination register is the same as the first source register.

When we describe the details of the ARM and x86 instruction sets in Chapters 4 and 5, we shall see many more examples of instructions that have different address formats. We will be able to appreciate the tradeoffs of having different address formats in all their glory.

3.2.5 Types of Operands

Let us now look at the different types of operands. The method of specifying and accessing an operand in an assembly statement is known as the **addressing mode**.

Definition 32
The method of specifying and accessing an operand in an assembly statement is known as the addressing mode.

The simplest way of specifying an operand is by embedding its value in the instruction. Most assembly languages allow the user to specify the values of integer constants as an operand. This addressing mode is known as the *immediate* addressing mode. This method is very useful for initialising registers or memory locations, or for performing arithmetic operations.

Once the requisite set of constants have been loaded into registers and memory locations, the program needs to proceed by operating on registers and memory locations. There are several addressing modes in this space. Before introducing them, let us introduce some extra terminology in the form of the *register transfer notation*.

Register Transfer Notation

This notation allows us to specify the semantics of instructions and operands. Let us look at the various methods to represent the basic actions of instructions.

$$r1 \leftarrow r2$$

This expression has two register operands $r1$, and $r2$. $r1$ is the destination register, and $r2$ is the source register. We are transferring the contents of register $r2$ to register $r1$.

We can specify an add operation with a constant as follows:

$$r1 \leftarrow r2 + 4$$

We can also specify operations on registers using this notation. We are adding the contents of $r2$ and $r3$ and saving the result in $r1$.

$$r1 \leftarrow r2 + r3$$

It is also possible to represent memory accesses using this notation.

$$r1 \leftarrow [r2]$$

In this case the memory address is saved in $r2$. The processor hardware fetches the memory address in $r2$, accesses the location, fetches the contents of the memory location, and saves the data item in $r1$. Let us assume that the value in $r2$ is 100. In this case the processor accesses memory with address 100, fetches the integer saved in locations (100-103), and saves it in $r1$. By default we assume that we are loading and saving integers.

We can also specify a more complicated memory address of the form:

$$r1 \leftarrow [r2 + 4]$$

Here, the memory address is equal to the contents of the register $r2$ plus 4. We fetch the integer starting at the contents of this memory address, and save it in the register $r1$.

Generic Addressing Modes for Operands

Let us represent the value of an operand as V. In the subsequent discussion, we use expressions such as $V \leftarrow r1$. This does not mean that we have a new storage location called V. It basically means that the value of an operand is specified by the RHS (right hand side). Let us briefly take a look at some of the most commonly used addressing modes with examples.

immediate $V \leftarrow imm$
Uses the constant imm as the value of the operand.

register $V \leftarrow r1$
In this addressing mode, the processor uses the value contained in a register as the operand.

register-indirect $V \leftarrow [r1]$
The register saves the address of the memory location that contains the value.

base-offset $V \leftarrow [r1 + offset]$
$offset$ is a constant. The processor fetches the base memory address from $r1$, adds the constant $offset$ to it, and accesses the new memory location to fetch the value of the operand. The $offset$ is also known as the $displacement$.

base-index $V \leftarrow [r1 + r2]$
$r1$ is the base register, and $r2$ is the index register. The memory address is equal to $(r1 + r2)$.

base-index-offset $V \leftarrow [r1 + r2 + offset]$
The memory address that contains the value is $(r1 + r2 + offset)$, where $offset$ is a constant.

memory-direct $V \leftarrow addr$
The value is contained in memory starting from address $addr$. $addr$ is a constant. In this case the memory address is directly embedded in the instruction.

memory-indirect $V \leftarrow [[r1]]$
The value is present in a memory location, whose address is contained in the memory location, M. Furthermore, the address of M is contained in the register, $r1$.

PC-relative $V \leftarrow [PC + offset]$
Here, $offset$ is a constant. The memory address is computed to be $PC + offset$, where PC represents the value contained in the PC. This addressing mode is useful for branch instructions.

Let us introduce a new term called the *effective memory address* by considering the base-offset addressing mode. The memory address is equal to the contents of the base register plus the offset. The computed memory address is known as the *effective memory address*. We can similarly define the effective address for other addressing modes in the case of memory operands.
.

Definition 33
The memory address specified by an operand is known as the effective memory address.

3.3 *SimpleRisc*

In this book, we shall introduce a simple, generic, complete and concise RISC ISA called *SimpleRisc* . The assembly language of *SimpleRisc* has just 21 instructions, and captures most of the features of full scale assembly languages. We will use *SimpleRisc* to demonstrate the flavour of different types of assembly programs, and also design a processor for the *SimpleRisc* ISA in Chapter 8. We shall assume that *SimpleRisc* assembly follows the GNU assembly format, and we shall only describe the text section in this book.

Before proceeding further, let us take a tour of different instruction sets, and take a look at their properties.

3.3.1 Different Instruction Sets

In Chapter 1, we looked at properties of different instruction sets including necessary, and desirable properties. In this book, we shall describe two real instruction sets namely the *ARM* instruction set and x86 instruction set. ARM stands for "Advanced RISC Machines". It is an iconic company based out of Cambridge, UK. As of 2012, around 90% of mobile devices including the Apple iPhone, and iPad, run on ARM based processors. Similarly, as of 2012, more than 90% of the desktops and laptops run on Intel or AMD based x86 processors. ARM is a RISC instruction set, and x86 is a CISC instruction set.

There are many other instruction sets tailored for a wide variety of processors. Another popular instruction set for mobile computers is the MIPS instruction set. MIPS based processors are also used in a wide variety of processors used in automobiles, and industrial electronics.

ISA	Type	Year	Vendor	Bits	Endianness	Registers
VAX	CISC	1977	DEC	32	little	16
SPARC	RISC	1986	Sun	32	big	32
	RISC	1993	Sun	64	bi	32
PowerPC	RISC	1992	Apple,IBM,Motorola	32	bi	32
	RISC	2002	Apple,IBM	64	bi	32
PA-RISC	RISC	1986	HP	32	big	32
	RISC	1996	HP	64	big	32
m68000	CISC	1979	Motorola	16	big	16
	CISC	1979	Motorola	32	big	16
MIPS	RISC	1981	MIPS	32	bi	32
	RISC	1999	MIPS	64	bi	32
Alpha	RISC	1992	DEC	64	bi	32
x86	CISC	1978	Intel,AMD	16	little	8
	CISC	1985	Intel,AMD	32	little	8
	CISC	2003	Intel,AMD	64	64 little	16
ARM	RISC	1985	ARM	32	bi (little default)	16
	RISC	2011	ARM	64	bi (little default)	31

Table 3.1: List of instruction sets

For large servers, typically IBM (PowerPC), Sun (now Oracle)(UltraSparc), or HP (PA-RISC) processors are used. Each of these processor families has its own instruction set. These instruction sets are typically RISC instruction sets. Most ISAs share simple instructions such as add, subtract, multiply, shifts, and load/store instructions. However, beyond this simple set, they use a large number of more specialised instructions. As we shall see in the next few chapters, choosing the right set of instructions in an ISA is dependent on the target market of the processor, the nature of the workload, and many design time constraints. Table 3.1 shows a list of popular instruction sets. The *SimpleRisc* ISA is conceptually the closest to ARM and MIPS; however, it has some significant differences also.

3.3.2 Model of the ***SimpleRisc*** Machine

SimpleRisc assumes that we have 16 registers numbered $r0 \dots r15$. The first 14 registers are general purpose registers, and can be used for any purpose within the program. Register $r14$ is known as the stack pointer. We shall also refer to it as *sp*. Register $r15$ is known as the return address register, and it will also be referred as ra. We shall discuss *sp* and ra, when we discuss how to implement functions in *SimpleRisc* . Each register is 32 bits wide. We assume a special internal register called $flags$, which is not visible to the programmer. It contains two fields $flags.E$(equal) and $flags.GT$(greater than). E is set to 1 if the result of a comparison is equality, and GT is set to 1 if a comparison concludes that the first operand is greater than the second operand. The default values of both the fields are 0.

Each instruction is encoded into a 32-bit value, and it requires 4 bytes of storage in memory.

SimpleRisc assumes a memory model similar to the Von Neumann machine augmented with registers as described in Section 1.7.3. The memory is a large array of bytes. A part of it saves the program and the rest of the memory is devoted to storing data. We assume that multibyte data types such as integers are saved in the little endian format.

3.3.3 Register Transfer Instruction – *mov*

The *mov* instruction is a 2-address format instruction that can transfer values from one register to another, or can load a register with a constant. Our convention is to always have the destination register at the beginning. Refer to Table 3.2. The size of the signed immediate operand is limited to 16 bits. Hence, its range is between -2^{15} to $2^{15} - 1$.

Semantics	Example	Explanation
mov *reg*, (*reg*/*imm*)	mov r1, r2	$r1 \leftarrow r2$
	mov r1, 3	$r1 \leftarrow 3$

Table 3.2: Semantics of the *mov* instruction

3.3.4 Arithmetic Instructions

SimpleRisc has 6 arithmetic instructions – *add*, *sub*, *mul*, *div*, *mod*, and *cmp*. The connotations of *add*, *sub*, and *mul* are self explanatory (also see Table 3.3). For arithmetic instructions, we assume that the first operand in the list of operands is the destination register. The second

operand is the first source operand, and the third operand is the second source operand. The first and second operands need to be registers, whereas the last operand (second source register) can be an immediate value.

Semantics	Example	Explanation
add *reg*, *reg*, (*reg*/*imm*)	add r1, r2, r3	$r1 \leftarrow r2 + r3$
	add r1, r2, 10	$r1 \leftarrow r2 + 10$
sub *reg*, *reg*, (*reg*/*imm*)	sub r1, r2, r3	$r1 \leftarrow r2 - r3$
mul *reg*, *reg*, (*reg*/*imm*)	mul r1, r2, r3	$r1 \leftarrow r2 \times r3$
div *reg*, *reg*, (*reg*/*imm*)	div r1, r2, r3	$r1 \leftarrow r2/r3$ (quotient)
mod *reg*, *reg*, (*reg*/*imm*)	mod r1, r2, r3	$r1 \leftarrow r2\ mod\ r3$ (remainder)
cmp *reg*, (*reg*/*imm*)	cmp r1, r2	set flags

Table 3.3: Semantics of arithmetic instructions in *SimpleRisc*

Example 26
*Write assembly code in SimpleRisc to compute: 31 * 29 - 50, and save the result in r4.*
Answer:

SimpleRisc
```
mov r1, 31
mov r2, 29
mul r3, r1, r2
sub r4, r3, 50
```

The *div* instruction divides the first source operand by the second source operand, computes the quotient, and saves it in the destination register. For example it will compute 30/7 to be 4. The *mod* instruction computes the remainder of a division. For example, it will compute 30 *mod* 7 as 2.

Example 27
Write assembly code in SimpleRisc to compute: 31 / 29 - 50, and save the result in r4.
Answer:

SimpleRisc
```
mov r1, 31
mov r2, 29
div r3, r1, r2
sub r4, r3, 50
```

The *cmp* instruction is a 2-address instruction that takes two source operands. The first source operand needs to be a register, and the second one can be an immediate or a register. It

compares both the operands by subtracting the second from the first. If the operands are equal, or in other words the result of the subtraction is zero, then it sets $flags.E$ to 1. Otherwise $flags.E$ is set to 0. If the first operand is greater than the second operand, then the result of the subtraction will be positive. In this case, the *cmp* instruction sets $flags.GT$ to 1, otherwise it sets it to 0. We will require these flags when we implement branch instructions.

3.3.5 Logical Instructions

SimpleRisc has three logical instructions – *and*, *or*, and *not*. *and* and *or* are 3-address instructions. They compute the bitwise AND and OR of two values respectively. The *not* instruction is a 2-address instruction that computes the bitwise complement of a value. Note that the source operand of the *not* instruction can be an immediate or a register. Refer to Table 3.4.

Semantics	Example	Explanation
and *reg*, *reg*, (*reg*/*imm*)	and r1, r2, r3	$r1 \leftarrow r2 \wedge r3$
or *reg*, *reg*, (*reg*/*imm*)	or r1, r2, r3	$r1 \leftarrow r2 \vee r3$
not *reg*, (*reg*/*imm*)	not r1, r2	$r1 \leftarrow \sim r2$
$\wedge$ bitwise AND, $\vee$ bitwise OR, $\sim$ logical complement		

Table 3.4: Semantics of logical instructions in *SimpleRisc*

Example 28
Compute $\overline{(a \vee b)}$. *Assume that* a *is stored in* $r0$, *and* b *is stored in* $r1$. *Store the result in* $r2$.
Answer:

SimpleRisc
```
or r3, r0, r1
not r2, r3
```

3.3.6 Shift Instructions – *lsl*, *lsr*, *asr*

SimpleRisc has three types of shift instructions *lsl* (logical shift left), *lsr* (logical shift right), and *asr* (arithmetic shift right). Each of these instructions are in the 3-address format. The first source operand points to the source register, and the second source operand contains the shift amount. The second operand can either be a register or an immediate value.

The *lsl* instruction shifts the value in the first source register to the left Similarly, *lsr*, shifts the value in the first source register to the right. Note that it is a logical right shift. This means that it fills all the MSB positions with zeros. In comparison, *asr*, performs an arithmetic right shift. It fills up all the MSB positions with the value of the previous sign bit. Semantics of shift instructions are shown in Table 3.5.

Semantics	Example	Explanation
lsl *reg*, *reg*, (*reg*/*imm*)	lsl r3, r1, r2	$r3 \leftarrow r1 \ll r2$ (shift left)
	lsl r3, r1, 4	$r3 \leftarrow r1 \ll 4$ (shift left)
lsr *reg*, *reg*, (*reg*/*imm*)	lsr r3, r1, r2	$r3 \leftarrow r1 \ggg r2$ (shift right logical)
	lsr r3, r1, 4	$r3 \leftarrow r1 \ggg 4$ (shift right logical)
asr *reg*, *reg*, (*reg*/*imm*)	asr r3, r1, r2	$r3 \leftarrow r1 \gg r2$ (arithmetic shift right)
	asr r3, r1, 4	$r3 \leftarrow r1 \gg 4$ (arithmetic shift right)

Table 3.5: Semantics of shift instructions in *SimpleRisc*

3.3.7 Data Transfer Instructions: *ld* and *st*

SimpleRisc has two data transfer instructions – load(*ld*) and store(*st*). The load instructions loads values from memory into registers, and the store instruction saves values in registers to memory locations. Examples and semantics are shown in Table 3.6.

Semantics	Example	Explanation
ld *reg*, *imm*[*reg*]	ld r1, 12[r2]	$r1 \leftarrow [r2 + 12]$
st *reg*, *imm*[*reg*]	st r1, 12[r2]	$[r2 + 12] \leftarrow r1$

Table 3.6: Semantics of load-store instructions in *SimpleRisc*

Let us consider the load instruction: *ld* $r1, 12[r2]$. Here, we are computing the memory address as the sum of the contents of $r2$ and the number 12. The *ld* instructions accesses this memory address, fetches the stored integer and stores it in $r1$. We assume that the computed memory address points to the first stored byte of the integer. Since we assume a little endian representation, the memory address contains the LSB. The details are shown in Figure 3.7(a).

The store operation does the reverse. It stores the value of $r1$ into the memory address (r2 + 12). Refer to Figure 3.7(b).

3.3.8 Unconditional Branch Instructions

SimpleRisc has one unconditional branch instruction, *b*, which makes the program counter jump to the address corresponding to a label in the code. It takes a single operand, which is a label in the program. Its semantics is shown in Table 3.7.

Semantics	Example	Explanation
b *label*	b .foo	branch to .foo

Table 3.7: Semantics of unconditional branch instructions in *SimpleRisc*

Let us explain its operation with the help of a simple example, as shown below.

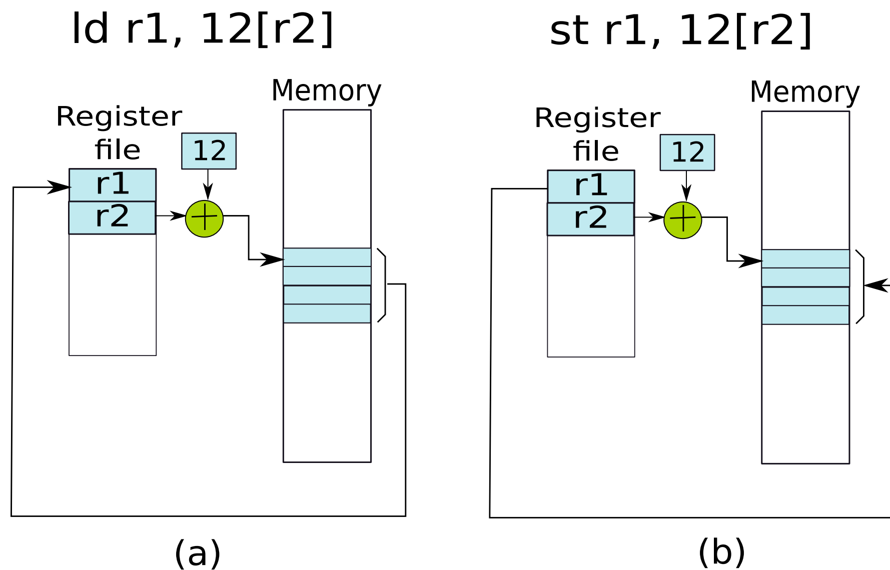


Figure 3.7: Load and store operations in *SimpleRisc*

```
add r1, r2, r3
b .foo
...
...
.foo:
        add r3, r1, r4
```

In this example, we add the values of $r2$, and $r3$, and then save the result in $r1$. After that, the processor jumps to the code pointed to by the label, $.foo$. It proceeds to execute the code after the label, $.foo$. It starts out by executing the instruction *add* $r3, r1, r4$. It then proceeds to execute subsequent instructions.

3.3.9 Conditional Branch Instructions

SimpleRisc has two conditional branch instructions – *beq* and *bgt*. Real world instruction sets typically have more branch instructions. Nonetheless, at the cost of code size, these two instructions are sufficient for implementing all types of branches.

The *beq* instruction stands for "branch if equal". This means that if any preceding *cmp* instruction has set the E flag, then the PC will branch to the label specified in this instruction. Otherwise, the branch is said to fail, and the processor will proceed to execute the instruction after the branch. Similarly, the *bgt* instruction stands for "branch if greater than". This branch instruction bases its outcome on the value of the GT flag. It if is set to 1, then it branches to the label specified in the branch instruction, otherwise the processor executes the next instruction after the branch. Refer to Table 3.8.

Semantics	Example	Explanation
beq *label*	beq .foo	branch to .foo if $flags.E = 1$
bgt *label*	bgt .foo	branch to .foo if $flags.GT = 1$

Table 3.8: Semantics of ranch instructions in *SimpleRisc*

Example 29
Write an iterative program to compute the factorial of a number stored in r0. Assume that the number is greater than 2. Save the result in r1.
Answer: *Let us first take a look at a small C program to compute the factorial of the variable num.*

C
```
int prod = 1;
int idx;
for(idx = num; idx > 1; idx --) {
        prod = prod * idx
}
```

Let us now try to convert this program to SimpleRisc .

SimpleRisc
```
mov r1, 1           /* prod = 1 */
mov r2, r0          /* idx = num */
.loop:
        mul r1, r1, r2  /* prod = prod * idx */
        sub r2, r2, 1   /* idx = idx - 1 */
        cmp r2, 1       /* compare (idx, 1) */
        bgt .loop       /* if (idx > 1) goto .loop*/
```

Example 30 *Write an assembly program to find out if the number stored in r1 is a prime number. Assume that it is greater than 3. Save the Boolean result in r0.*
Answer:

SimpleRisc
```
    mov r2, 2
.loop:
    mod r3, r1, r2  @ divide number by r2
    cmp r3, 0       @ compare the result with 0
    beq .notprime   @ if the result is 0, not prime
    add r2, r2, 1   @ increment r2
    cmp r1, r2      @ compare r2 with the number
```

```
    bgt .loop       @ iterate if r2 is smaller

mov r0, 1           @ number is prime
b .exit             @ exit

.notprime:
    mov r0, 0       @ number is not prime

.exit:
```

Example 31 *Write an assembly program to find the least common multiple (LCM) of two positive numbers stored in r1 and r2. Save the result in r0.*

Answer:

SimpleRisc
```
@ let the numbers be A(r1) and B(r2)

@ iterate
    mov r3, 1           @ idx = 1
    mov r4, r1          @ L = A

.loop:
    mod r5, r4, r2      @ tmp = L % b
    cmp r5, 0           @ compare mod with 0
    beq .lcm            @ LCM found (L is the LCM)
    add r3, r3, 1       @ increment idx
    mul r4, r1, r3      @ L = A * idx
    b .loop

.lcm:
    mov r0, r4          @ result is equal to L
```

3.3.10 Functions

Now, that we have seen generic instructions, operands, and addressing modes, let us come to one of the most advanced features in high level programming languages that makes their structure extremely modular namely *functions* (also referred to as subroutines or procedures in some languages). If the same piece of code is used at different points in a program, then it can be encapsulated in a function. The following example shows a function in C to add two numbers.

```
int addNumbers(int a, int b) {
        return (a+b);
}
```

Calling and Returning from Functions

Let us now go over the basic requirements to implement a simple function. Let us assume that an instruction with address A *calls* a function foo. After executing function foo, we need to come back to the instruction immediately after the instruction at A. The address of this instruction is $A + 4$ (if we assume that the instruction at A is 4 bytes long). This process is known as *returning from a function*, and the address $(A + 4)$ is known as the *return address*.

> **Definition 34**
> *Return address: It is the address of the instruction that a process needs to branch to after executing a function.*

Thus, there are two fundamental aspects of implementing a function. The first is the process of invoking or calling a function, and the second aspect deals with returning from a function.

Let us consider the process of calling a function in bit more detail. A function is essentially a block of assembly code. Calling a function is essentially making the PC point to the start of this block of code. We have already seen a method to implement this functionality when we discussed branch instructions. We can associate a label with every function. The label should be associated with the first instruction in a function. Calling a function is as simple as branching to the label at the beginning of a function. However, this is only a part of the story. We need to implement the return functionality as well. Hence, we cannot use an unconditional branch instruction to implement a function call.

Let us thus propose a dedicated function call instruction that branches to the beginning of a function, and simultaneously saves the address that the function needs to return to (referred to as the return address). Let us consider the following C code, and assume that each C statement corresponds to one line of assembly code.

```
a = foo(); /* Line 1 */
c = a + b; /* Line 2 */
```

In this small code snippet, we use a function call instruction to call the foo function. The return address is the address of the instruction in Line 2. It is necessary for the call instruction to save the return address in a dedicated storage location such that it can be retrieved later. Most RISC instruction sets (including *SimpleRisc*) have a dedicated register known as the return address register to save the return address. The return address register gets automatically populated by a function *call* instruction. When we need to return from a function, we need to branch the address contained in the return address register. In *SimpleRisc* , we devote register 15 to save the return address, and refer to it as ra.

What happens if foo calls another function? In this case, the value in ra will get overwritten. We will look at this issue later. Let us now consider the problem of passing arguments to a function, and getting return values back.

Passing Arguments and Return Values

Assume that a function foo invokes a function $foobar$. foo is called the *caller*, and $foobar$ is called the *callee*. Note that the caller-callee relationships are not fixed. It is possible for foo to call $foobar$, and also possible for $foobar$ to call foo in the same program. The caller and callee are decided for a single function call based on which function is invoking the other.

Definition 35

caller *A function, foo, that has called another function, $foobar$.*

callee *A function, $foobar$, that has been called by another function, foo.*

Both the caller and the callee see the same view of registers. Consequently, we can pass arguments through the registers, and likewise pass the return values through registers also. However, there are several issues in this simple idea as we enumerate below (Assume that we have 16 registers).

1. A function can take more than 16 arguments. This is more than the number of general purpose registers that we have. Hence, we need to find a extra space to save the arguments.
2. A function can return a large amount of data, for example, a large structure in C. It might not be possible for this piece of data to fit in registers.
3. The callee might overwrite registers that the caller might require in the future.

We thus observe that passing arguments and return values through registers works only for simple cases. It is not a very flexible and generic solution. Nonetheless, there are two requirements that emerge from our discussion.

Space Problem We need extra space to send and return more arguments.

Overwrite Problem We need to ensure that the callee does not overwrite the registers of the caller.

To solve both the problems, we need to take a deeper look at how functions really work. We can think of a function – foo – as a black box to begin with. It takes a list of arguments and returns a set of values. To perform its job, foo can take one nano-second, or one week, or even one year. foo might call other functions to do its job, send data to I/O devices, and access memory locations. Let us visualise the function, foo, in Figure 3.8.

To summarise, a generic function processes the arguments, reads and writes values from memory and I/O devices if required, and then returns the result. Regarding memory and I/O devices, we are not particularly concerned at this point of time. There is a large amount of memory available, and space is not a major constraint. Reading and writing I/O devices is also

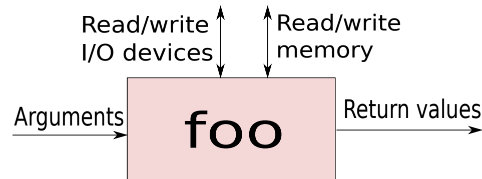


Figure 3.8: Function foo as a black box.

typically not associated with space constraints. The main issue is with registers, because they are in short supply.

Let us solve the **space problem** first. We can transfer values through both registers and memory. For simplicity, if we need to transfer a small amount of data, we can use registers, otherwise we can transfer them through memory. Similarly, for return values, we can transfer values through memory. We are not limited by space constraints if we use memory to transfer data. However, this approach suffers from lack of flexibility. This is because there has to be strict agreement between the caller and the callee regarding the memory locations to be used. Note that we cannot use a fixed set of memory locations, because it is possible for the callee to recursively call itself.

recursive function call

```
foobar() {
    ...
    foobar();
    ...
}
```

An astute reader might argue that it is possible for the callee to read the arguments from memory and transfer them to some other temporary area in memory and then call other functions. However, such approaches are not elegant and not very efficient also. We shall look at more elegant solutions later.

Hence, at this point, we can conclude that we have solved the space problem partially. If we need to transfer a few values between the caller and the callee or vice versa, we can use registers. However, if the arguments/return values do not fit in the set of available registers, then we need to transfer them through memory. For transferring data through memory, we need an elegant solution that does not require a strict agreement between the caller and the callee regarding the memory locations used to transfer data. We shall consider such solutions in Section 3.3.10.

Definition 36
The notion of saving registers in memory and later restoring them is known as register spilling.

To solve the **overwrite problem**, there are two solutions. The first is that the caller can save the set of registers it requires in a dedicated location in memory. It can later retrieve its set of registers after the callee finishes, and returns control to the caller. The second solution is for the callee to save and restore the registers that it will require. Both the approaches are shown in Figure 3.9. This method of saving the values of registers in memory, and later retrieving them is known as *spilling*.

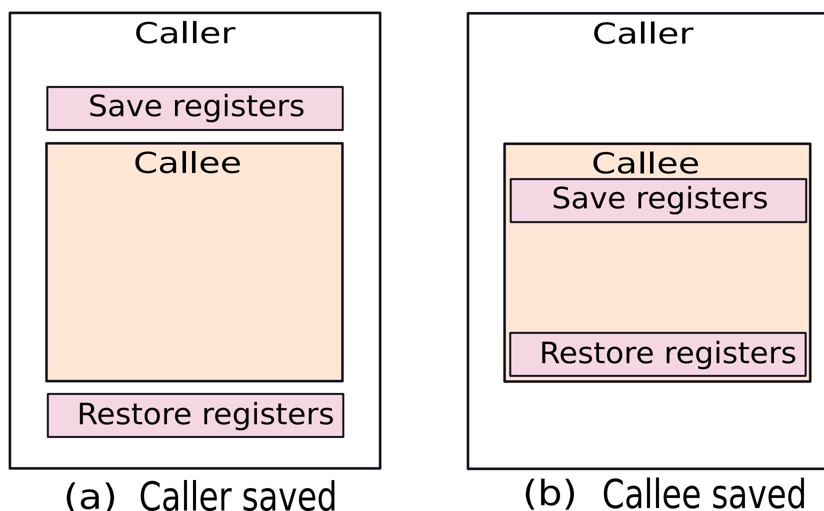


Figure 3.9: Caller saved and callee saved registers

Here, we have the same problem again. Both the caller and the callee need to have a strict agreement on the locations in memory that need to be used. Let us now try to solve both the problems together.

The Stack

We simplified the process of passing arguments to and from a function, and saving/restoring the registers using dedicated locations in memory. However, this solution was found to be inflexible and it can be quite complex to implement for large real world programs. To simplify this idea, let us find a pattern in function calls.

A typical C or Java program starts with the *main* function. This function then calls other functions, which might in turn call other functions, and finally the execution terminates when the main function exits. Each function defines a set of local variables and performs a computation on these variables and the function arguments. It might also call other functions. Finally, the function returns a value and rarely a set of values (structure in C). Note that after a function terminates, the local variables, and the arguments are not required anymore. Hence, if some of these variables or arguments were saved in memory, we need to reclaim the space. Secondly, if the function has spilled registers, then these memory locations also need to be freed after it exits. Lastly, we note that if the callee calls another function, then it will need to save the value of the return address register in memory. We will need to free this location also after the function exits.

It is best to save all of these pieces of information contiguously in a single region of memory. This is known as the *activation block* of the function. Figure 3.10 shows the memory map of the activation block.

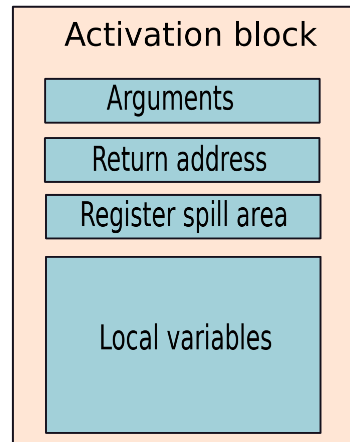


Figure 3.10: Activation block

The activation block contains the arguments, return address, register spill area (for both caller saved and callee saved schemes), and the local variables. Once a function terminates, it is possible to get rid of the activation block entirely. If a function wants to return some values, then it can either do so using registers. However, if it wants to return a large structure, then it can write it into the activation block of the caller. The caller can supply a location within its activation block where this data can be written. We shall see that it is possible to do this more elegantly. Prior to explaining how this can be done, we need to look at how to arrange activation blocks in memory.

We can have one memory region where all the activation blocks are stored in contiguous regions. Let us consider an example. Let us assume that function foo calls function $foobar$, which in turn calls $foobarbar$. Figure 3.11(a) - (d) show the state of memory at four points – (a) just before calling $foobar$, (b) just before calling $foobarbar$, (c) after calling $foobarbar$, (d) just after $foobarbar$ returns.

We observe that there is a last in first out behavior in this memory region. The function that was invoked the last is the first function to finish. Such kind of a last in-first out structure is traditionally known as a *stack* in computer science. Hence, the memory region dedicated to saving activation blocks is known as the *stack*. Traditionally, the stack has been considered to be downward growing (growing towards smaller memory addresses). This means that the activation block of the main function starts at a very high location and new activation blocks are added just below (towards lower addresses) existing activation blocks. Thus the top of the stack is actually the smallest address in the stack, and the bottom of the stack is the largest address. The top of the stack represents the activation block of the function that is currently executing, and the bottom of the stack represents the initial main function.

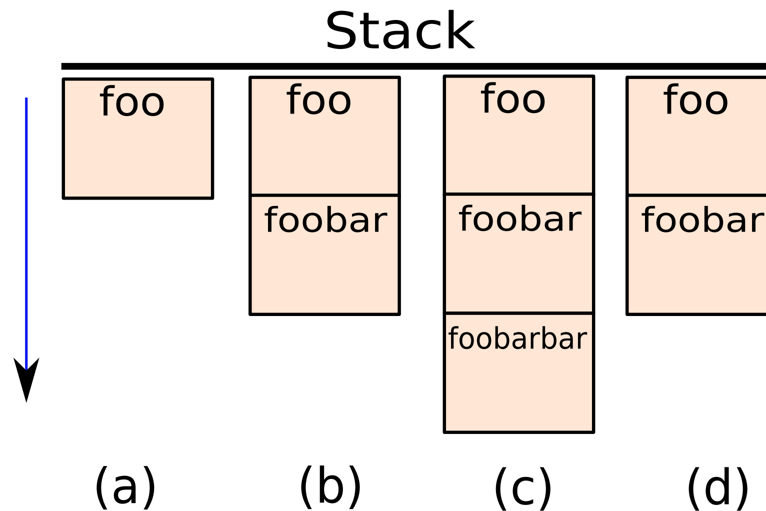


Figure 3.11: The state of the stack after several function calls

Definition 37
The stack *is a memory region that saves all the activation blocks in a program.*

- *It is traditionally considered to be downward growing.*
- *Before calling a function, we need to push its activation block to the stack.*
- *When a function finishes execution, we need to pop its activation block off the stack.*

Definition 38
The stack pointer register maintains a pointer to the top of the stack.

Most architectures save a pointer to the top of the stack in a dedicated register called the *stack pointer*. This register is $r14$ in *SimpleRisc* . It is also called *sp*. Note that for a lot of architectures, the stack is a purely software structure. For them, the hardware is not aware of the stack. However, for some architectures such as x86, hardware is aware of the stack and uses it to push the return address or the values of other registers. However, even in this case the hardware is not aware of the contents of each activation block. The structure is decided by the assembly programmer or the compiler. In all cases, the compiler needs to explicitly add assembly instructions to manage the stack.

Creating a new activation block for the callee involves the following steps.

1. Decrement the stack pointer by the size of the activation block.

2. Copy the values of the arguments.

3. Initialise any local variables by writing to their corresponding memory locations if required.

4. Spill any registers (store to the activation block) if required.

It is necessary to destroy the activation block upon returning from a function. This can be trivially done by adding the size of the activation block to the stack pointer.

By using a stack, we have solved all of our problems. The caller and the callee cannot overwrite each other's local variables. The local variables are saved in the activation blocks, and two activation blocks do not overlap. Along with variables it is possible to stop the callee from overwriting the caller's registers by explicitly inserting instructions to save registers in the activation blocks. There are two methods of achieving this – caller-saved scheme and callee-saved scheme. Secondly, there is no need to have an explicit agreement regarding the memory area that will be used to pass arguments. The stack can be used for this purpose. The caller can simply push the arguments on the stack. These arguments will get pushed into the callee's activation block, and the callee can easily use them. Similarly, while returning from a function the callee can pass return values through the stack. It needs to first destroy its activation block by decrementing the stack pointer, and then it can push the return values on the stack. The caller will be aware of the semantics of the callee, and thus after the callee returns it can assume that its activation block has been effectively enlarged by the callee. The additional space is consumed by the return values.

3.3.11 Function Call/Return Instructions

SimpleRisc has two instructions for functions – *call* and *ret*. The *call* instructions takes a single argument – the label of the first instruction of the function. It transfers control to the label and saves the return address in register *ra*. The *ret* instructions transfers the contents of *ra* to the PC. It is a 0-address instruction because it does not require any operands. Table 3.9 shows the semantics of these instructions. In Table 3.9, we assume that the *address* method provides the address of the first instruction of the foo function. Secondly, the return address is equal to $PC + 4$ because we assume that each instruction is 4 bytes long. *call* and *ret* can be thought of as *branch* instructions because they change the value of the PC. However, they are not dependent on any condition such as the value stored in a register. Hence, these instructions can conceptually be considered to be *unconditional branch instructions*.

Semantics	Example	Explanation
call *label*	call .foo	$ra \leftarrow PC + 4$; $PC \leftarrow address(.foo)$;
ret	ret	$PC \leftarrow ra$

Table 3.9: Semantics of function call/return instructions in *SimpleRisc*

Example 32
Write a function in SimpleRisc that adds the values in registers $r0$, and $r1$, and saves the result in $r2$.
Answer:

SimpleRisc
```
.foo:
        add r2, r0, r1
        ret
```

Example 33
Write a function, foo, in SimpleRisc that adds the values in registers $r0$, and $r1$, and saves the result in $r2$. Then write another function that invokes this function. The invoking function needs to first set $r0$ to 3, $r1$ to 5, and then invoke foo. After foo returns, it needs to add 10 to the result of foo, and finally save the sum in $r3$.
Answer:

SimpleRisc
```
.foo:
        add r2, r0, r1
        ret

.main:
        mov r0, 3
        mov r1, 5
        call .foo
        add r3, r2, 10
```

Example 34
Write a recursive function to compute the factorial of 10 that is initially stored in $r0$. Save the result in $r1$.
Answer: *Let us first take a look at a small C program to compute the factorial of the variable num.*

C
```
int factorial(int num) {
        if (num <= 1) return 1;
        return num * factorial(num - 1);
}
void main() {
```

```
        int result = factorial(10);
}
```

Let us now try to convert this program to SimpleRisc .

SimpleRisc

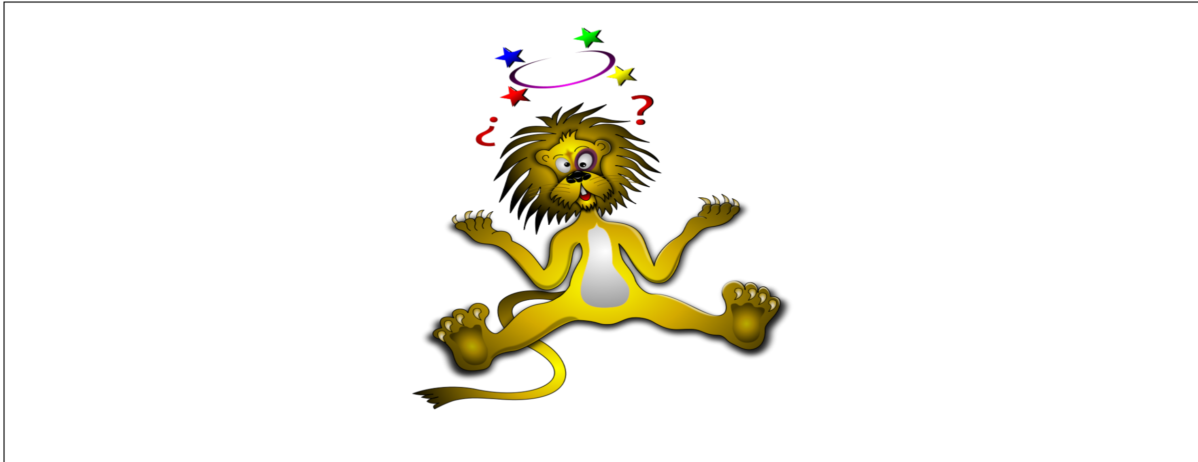
```
.factorial:
    cmp r0, 1       /* compare (1,num) */
    beq .return
    bgt .continue
    b .return

.continue:
    sub sp, sp, 8   /* create space on the stack */
    st r0, [sp]     /* push r0 on the stack */
    st ra, 4[sp]    /* push the return address register */
    sub r0, r0, 1   /* num = num - 1 */
    call .factorial /* result will be in r1 */
    ld r0, [sp]     /* pop r0 from the stack */
    ld ra, 4[sp]    /* restore the return address */
    mul r1, r0, r1  /* factorial(n) = n * factorial(n-1) */
    add sp, sp, 8   /* delete the activation block */
    ret
.return:
    mov r1, 1
    ret

.main:
    mov r0, 10
    call .factorial
```

This example uses the stack to save and restore the value of r0*. In this case, the caller saves and restores its registers.*

3.3.12 The *nop* Instruction

Let us now add an instruction called *nop* that does nothing. Unlike other instructions, we do not need a table explaining the semantics of the instruction, because it does absolutely nothing!!!

Question 4 *Why on earth would we add an instruction that does not do anything?*



We will justify the need to have a nop instruction in our portfolio of instructions in Chapter 9. We shall see that it is important to have an instruction that does not do anything to ensure correctness in execution. Let us for the time being bear with this extra instruction that does not seem to have any purpose. The reader will definitely appreciate the need for this instruction in Chapter 9, when we discuss pipelining.

3.3.13 Modifiers

Let us now consider the problem of loading a 32-bit constant into a register. The following code snippet shows us how to load the constant $0xFB12CDEF$.

```
/* load the upper two bytes */
mov r0, 0xFB12
lsl r0, r0, 16

/* load the lower two bytes with 0x CD EF */
mov r1, 0x CDEF
lsl r1, r1, 16
lsr r1, r1, 16 /* top 16 bits are zeros */

/* load all the four bytes */
add r0, r0, r1
```

This problem requires 6 instructions. The reader needs to note that loading constants is a common operation in programs. Hence, let us devise a mechanism to speedup the process, and load a constant in a register in two operations. Most assemblers provide directives to directly load constants. Nevertheless, these directives need to get translated into a basic sequence of assembly instructions. Thus directives do not fundamentally solve of our problem of loading constants into registers of memory locations efficiently.

We shall achieve this by using *modifiers*. Let us assign a modifier, 'u', or 'h', to an ALU instruction other than shift instructions. By default, we assume that when we load a 16-bit immediate into a 32-bit register, the processor automatically performs sign extension. This

means that it sets each of the 16 MSB bits to the sign of the immediate. This preserves the value of the immediate. For example, if our immediate is equal to -2, then its hexadecimal representation is 0x FF FE. If we try to store it in a register, then in effect, we are storing – 0x FF FF FF FE.

Let us have two additional modes. Let us add the suffix 'u' to an instruction to make it interpret the immediate as an unsigned number. For example, the instruction *movu r0, 0x FEAB*, will load 0x 00 00 FE AB into register r0. This suffix allows us to specify 16-bit unsigned immediate values. Secondly, let us add the suffix 'h' to an instruction to instruct it to load the 16-bit immediate into the upper half of a register. For example, *movh r0, 0x FEAB*, effectively loads 0x FE AB 00 00, into $r0$. We can use modifiers with all ALU instructions, with the exception of shift instructions.

Let us now consider the previous example of loading a 32-bit constant into a register. We can implement it with two instructions as follows:

```
movh r0, 0xFB12      /* r0 = 0xFB 12 00 00 */
addu r0, r0, 0xCDEF  /* r0 = r0 + 0x00 00 CD EF */
```

By using modifiers, we can load constants in 2 instructions, rather than 6 instructions. Furthermore, it is possible to create generic routines using modifiers that can set the value of any single byte in a 4 byte register. These routines will require a lesser number of instructions due to the use of modifiers.

3.3.14 Encoding the *SimpleRisc* Instruction Set

Let us now try to encode each instruction to a 32-bit value. We observe that we have instructions in 0,1,2 and 3 address formats. Secondly, some of the instructions take immediate values. Hence, we need to divide 32 bits into multiple fields. Let us first try to encode the type of instruction. Since there are 21 instructions, we require 5 bits to encode the instruction type. The code for each instruction is shown in Table 3.10. We can use the five most significant bits in a 32-bit field to specify the instruction type. The code for an instruction is also known as its *opcode*.

Definition 39
An opcode *is a unique identifier for each machine instruction.*

Now, let us try to encode each type of instruction starting from 0-address instructions.

Encoding 0-Address Instructions

The two 0-address instructions that we have are *ret*, and *nop*. The opcode is specified by the five most significant bits. In this case it is equal to 10100 for *ret*, and 10010 for *b* (refer to Table 3.10). Their encoding is shown in Figure 3.12. We only need to specify the 5 bit opcode in the MSB positions. The rest of the 27 bits are not required.

Instruction	Code	Instruction	Code	Instruction	Code
add	00000	not	01000	beq	10000
sub	00001	mov	01001	bgt	10001
mul	00010	lsl	01010	b	10010
div	00011	lsr	01011	call	10011
mod	00100	asr	01100	ret	10100
cmp	00101	nop	01101		
and	00110	ld	01110		
or	00111	st	01111		

Table 3.10: List of instruction opcodes

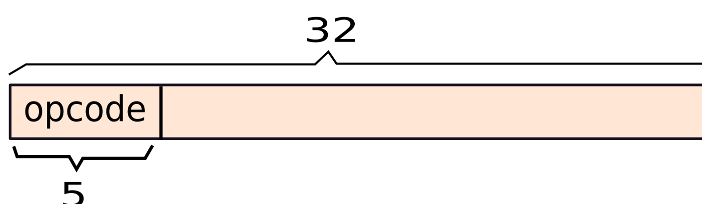


Figure 3.12: Encoding the *ret* instruction

Encoding 1-Address Instructions

The 1-address instructions that we have are *call*, *b*, *beq*, and *bgt*. In *SimpleRisc* assembly, they take a label as an argument. While encoding the instruction we need to specify the address of the label as the argument. The address of a label is the same as the address of the instruction that it is pointing to. If the line after the label is empty, then we need to consider the next assembly statement that has an instruction.

These four instructions require 5 bits for their opcode. The remaining 27 bits can be used for the address. Note that a memory address is 32 bits long. Hence, we cannot cover the address space with 27 bits. However, we can make two key optimisations. The first is that we can assume PC-relative addressing. We can assume that the 27 bits specify an offset (both positive and negative) with respect to the current PC. The branch statements in modern programs are generated because of *for/while* loops or *if-statements*. For these constructs the branch target is typically within a range of several hundred instructions. If we have 27 bits to specify the offset, and we assume that it is a 2's complement number, then the maximum offset in any direction (positive or negative) is 2^{26}. This is more than sufficient for almost all programs.

There is another important observation to be made. An instruction takes 4 bytes. If we assume that all instructions are aligned to 4-byte boundaries, then all starting memory addresses of instructions will be a multiple of 4. Hence, the least two significant binary digits of the address will be 00. There is no reason for wasting bits in trying to specify them. We can assume that the 27 bits specify the offset of the address of the memory word (in units of 4-byte memory words) that contains the instruction. With this optimisation, the offset from the PC in terms of bytes becomes 29 bits. This number should suffice for even the largest programs.

Just in case, there is a pathological example, in which the branch target is more than 2^{28} bytes away, then the assembler needs to chain the branches such that one branch will call another branch and so on. However, this would be a very rare case. The encoding for these instructions is shown in Figure 3.13.

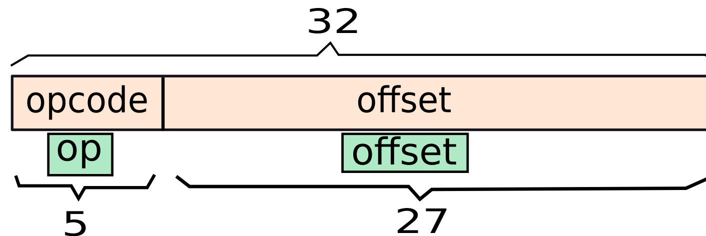


Figure 3.13: Encoding of 1-address instructions(*branch* format)

Note that the 1-address instruction format finds a use for the unused bits in the 0-address format. We can think of the 0-address format for the *ret* instruction as a special case of the 1-address format. Let us refer to the 1-address format as the *branch* format. Let us name the fields in this format. Let us call the opcode portion of the format as *op*, and the offset as $offset$. The *op* field contains the bits in positions 28-32, and the offset field contains the bits in positions 1-27.

Encoding 3-Address Instructions

Let us consider 3-address instructions first, and then look at other types of instructions. The 3-address instructions in *SimpleRisc* are *add*, *sub*, *mul*, *div*, *mod*, *and*, *or*, *lsl*, *lsr*, and *asr*.

Let us consider a generic 3-address instruction. It has a destination register, one input source register, and a second source operand that can either be a register or an immediate. We need to devote one bit to find out if the second source operand is a register or an immediate. Let us call this the I bit and specify it just after the opcode in the instruction. If $I = 1$, then the second source operand is an immediate. If $I = 0$, the second source operand is a register.

Let us now consider the case of 3-address registers that have their second source operand as a register($I = 0$). Since we have 16 registers, we require 4 bits to uniquely specify each register. Register ri can be encoded as the unsigned 4-bit binary equivalent of i. Hence, to specify the destination register and two input source registers, we require 12 bits. The structure is shown in Figure 3.14. Let us call this instruction format as the *register* format. Like the *branch* format let us name the different fields – *op* (opcode, bits: 28-32), I (immediate present, bits:27), rd (destination register, bits: 23-26), $rs1$ (source register 1, bits: 19-22), and $rs2$ (source register 2, bits:15-18).

Now, if we assume that the second source operand is an immediate, then we need to set I to 1. Let us calculate the number of bits we have left for specifying the immediate. We have already devoted 5 bits for the opcode, 1 bit for the I bit, 4 bits for the destination register, and 4 bits for the first source register. In all, we have expended 14 bits. Hence, out of 32 bits, we are left with 18 bits, and we can use them to specify the immediate.

We propose to divide the 18 bits into two parts – 2 bits (modifier) + 16 bits (constant part

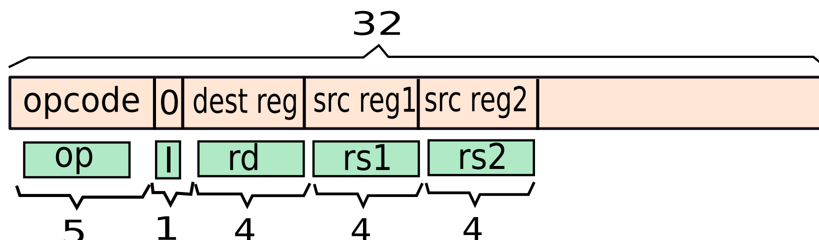


Figure 3.14: Encoding 3-address instructions with register operands (*register* format)

of the immediate). The two modifier bits can take three values – 00 (default), 01 ('u'), and 10 ('h'). The remaining 16 bits are used to specify a 16-bit 2's complement number when we are using default modifiers. For the *u* and *h* modifiers, we assume that the 16-bit constant in the immediate field is an unsigned number. In the rest of this book, we assume that the immediate field is 18 bits long with a modifier part, and a constant part. The processor internally expands the immediate to a 32-bit value, in accordance with the modifiers.

This encoding is shown in Figure 3.15. Let us call this instruction format as the *immediate* format. Like the *branch* format let us name the different fields – *op* (opcode, bits: 28-32), *I* (immediate present, bits:27), *rd* (destination register, bits: 23-26), *rs*1 (source register 1, bits: 19-22), and *imm* (immediate, bits:1-18).

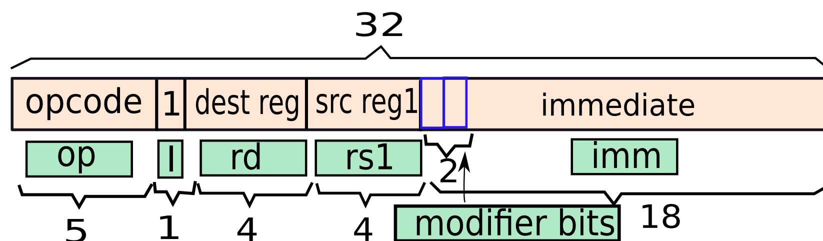


Figure 3.15: Encoding 3-address instructions with an immediate source operand (*immediate* format)

Example 35
Encode the instruction: sub r1, r2, 3.

Answer: *Let us encode each field of the instruction. We have:*

Field	*Encoding*
sub	*00001*
I	*1*
r1	*0001*
r2	*0010*
3	*11*

Thus, the binary encoding is (spaces added for readability): 00001 1 0001 0010 00 0000 0000 0000 0011. When we convert to hex, we get: 0x0C480003.

Encoding *cmp*, *not*, and *mov*

The *cmp* instruction has two source operands. The second source operand can be a register or an immediate. We will use the standard 3-address *register* or *immediate* formats for encoding the *cmp* instruction. The destination register field will remain empty. See Figure 3.16. One of our aims in designing the encoding is to keep things as simple and regular as possible such that the processor can decode the instruction very easily. We could have designed a separate encoding for a 2-address instruction such as *cmp*. However, the gains would have been negligible, and by sticking to a fixed format, the processor's instruction decode logic becomes more straight forward.

The *not* and *mov* instructions have one destination register, and one source operand. This source operand can be either an immediate or a register. Hence, we can treat the source operand of these instructions as the second source operand in the 3-address format, and keep the field for the first source register empty for both of these instructions. The format is shown in Figure 3.16.

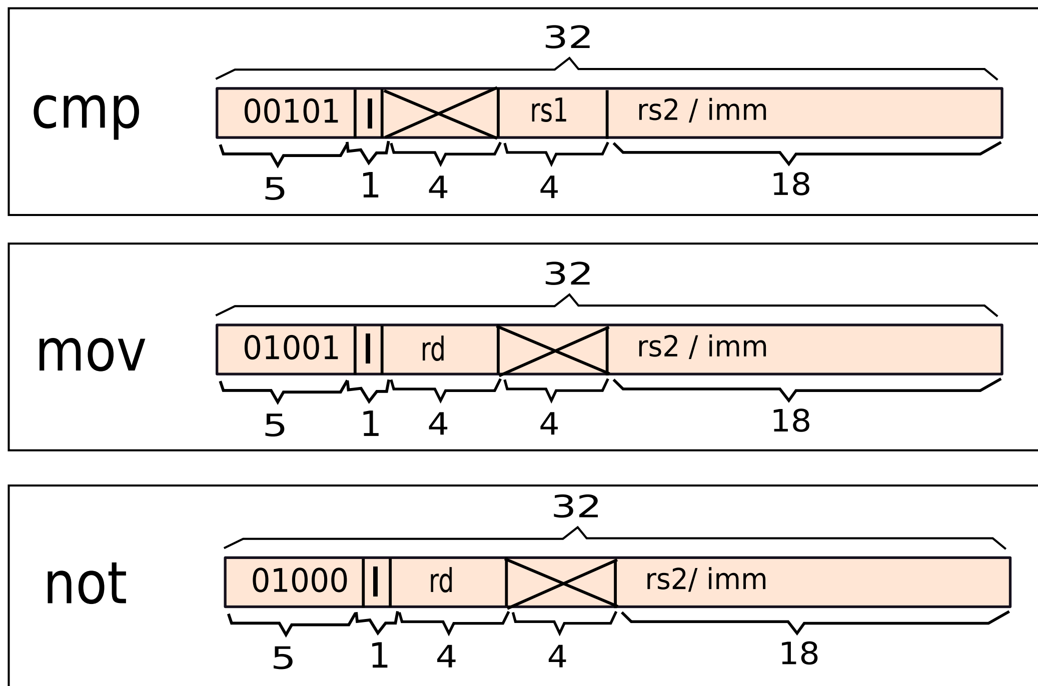


Figure 3.16: *cmp*, *not*, and *mov* instructions

Load and Store Instructions

In *SimpleRisc* the instructions – *ld* and *st* – are 2-address instructions. The second operand points to a memory address. It uses a base-offset addressing mode. There is a base register, and an integer offset.

For a load instruction, there are three unique pieces of information that need to be encoded: destination register, base register, and offset. In this case, we propose to use the three address *immediate* format. The *I* bit is set to 1, because we need to specify an offset. The first source register represents the base register, and the immediate represents the offset. Note that this encoding follows our principle of regularity and simplicity. Our aim is to reuse the 3-address *register* and *immediate* formats for as many instructions as possible.

Now, let us look at store instructions. Store instructions are slightly special in the sense that they do not have a destination register. The destination of a store instruction is a memory location. This information cannot be encoded in the *immediate* format. However, for reasons of simplicity, we still want to stick to the formats that we have defined. We need to take a crucial design decision here by answering Question 5.

Question 5
Should we define a new instruction format for the store instruction?

Let us adjudge this case in the favor of not introducing a new format. Let us try to reuse the *immediate* format. The *immediate* format has four fields – *op*, *rd*, $rs1$, and *imm*. The opcode field (*op*) need not be touched. We can assume that the format of the store instruction is: st rd, imm[rs1]. In this case, the field *rd* represents the register to be stored. Like the load instruction we can keep the base register as $rs1$, and use the *imm* field to specify the offset. We break the pattern we have been following up till now by saving a source register in *rd*, which is meant to save a destination register. However, we were compelled to do this at the cost of not introducing a new instruction format. Such design tradeoffs need to be made continuously. We have to always balance the twin objectives of elegance and efficiency. It is sometimes not possible to choose the best of both worlds. In this case, we have gone for efficiency, because introducing a new instruction format for just one instruction is overkill.

To conclude, figure 3.17 shows the encoding for load and store instructions.

Example 36
Encode the instruction: st r8, 20[r2].

Answer: *Let us encode each field of the instruction. We have:*

Field	*Encoding*
st	*01111*
I	*1*
r8	*1000*
r2	*0010*
20	*0001 0100*

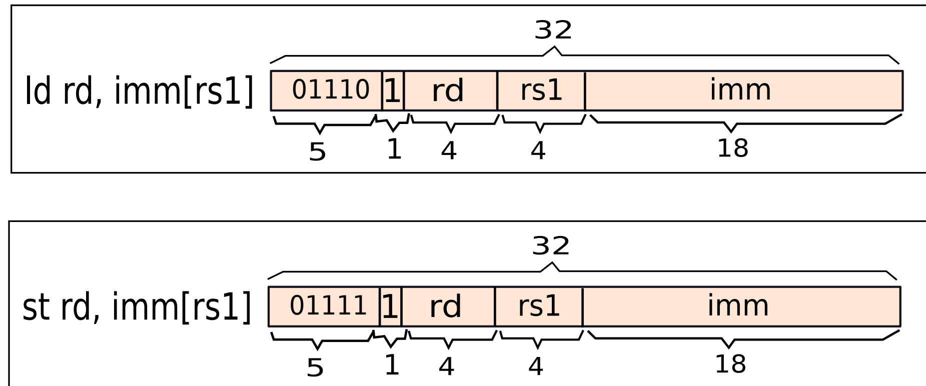


Figure 3.17: Encoding of load and store instructions

Thus, the binary encoding is (spaces added for readability): 01111 1 1000 0010 00 0000 0000 0001 0100. When we convert to hex, we get: 0x7E080014.

Summary of Instruction Formats

In the last few subsections, we have described a method to encode an instruction into a sequence of bits (machine code). A compiler can use this method to translate a program written in a high level language to machine code, and thus create an executable program. It is now the job of the processor to execute this program by reading the instructions one by one. We have substantially made our life easy by assuming that each instruction is exactly 4 bytes long. The processor simply needs to start at the starting address of the program in memory and fetch one instruction after the other. If an instruction is a branch, then the processor needs to evaluate the branch condition, and jump to the branch target. The part of the processor that is primarily concerned about the details of the ISA is the *decode* logic or the *decoder* . It is the role of the decoder to understand and decode an instruction. While designing an encoding for an ISA, creating a simple and efficient instruction decoder was our prime objective.

Format	Definition				
branch	op (28-32)	$offset$ (1-27)			
register	op (28-32)	I (27)	rd (23-26)	$rs1$ (19-22)	$rs2$ (15-18)
immediate	op (28-32)	I (27)	rd (23-26)	$rs1$ (19-22)	imm (1-18)
$op \rightarrow$ opcode, $offset \rightarrow$ branch offset, $I \rightarrow$ immediate bit, $rd \rightarrow$ destination register					
$rs1 \rightarrow$ source register 1, $rs2 \rightarrow$ source register 2, $imm \rightarrow$ immediate operand					

Table 3.11: Summary of instruction formats

To decode a *SimpleRisc* instruction, the first task is to find the instruction format. We have

defined three formats – *branch*, *immediate*, and *register*. Let us refer to Table 3.10. The six *branch* format instructions are *call*, *ret*, *beq*, *bgt*, *b*, and *nop*. Recall that we encode both 0 and 1-address format instructions in the *branch* format.

The opcodes of all the five branch instructions (*b*, *beq*, *bgt*, *call*, *ret*) have 1 as their most significant bit, whereas all other instructions have a 0 in their most significant position. Hence, for a decoder to find out if an instruction is a branch is very easy. It just needs to take a look at the three most significant bit of the opcode. It should be 1. Moreover, to find out if an instruction is a *nop*, the decoder needs to compare it with 01101, which requires a small circuit.

If an instruction is not in the *branch* format, then it must be in the *immediate* or *register* format. This can be quickly decided by taking a look at the *I* bit. If it is 1, then the instruction is in the *immediate* format, otherwise it is in the *register* format. The formats are summarised in Table 3.11.

Lessons Learnt

Now that we have designed a small instruction set of our own, looked at sample programs, and encoded our instructions, we are all set to design a processor for our *SimpleRisc* ISA. It needs to decode every single instruction, and execute it accordingly. Before proceeding further, let us look back at how we designed our ISA, and how should ISAs be designed in general.

1. The first step in designing an ISA is to study the workload that the ISA is being designed for. In the case of *SimpleRisc* , we wanted to use it for running general purpose programs. This meant that *SimpleRisc* needed to be simple, concise, generic, and complete as outlined in Chapter 1. However, for different target workloads, the requirements might be very different.

2. After studying the workload, we need to next decide on the number of instructions that we need to have. Unless there are compelling requirements otherwise, it is not advisable to have more than 64-128 instructions. More than 128 instructions will make the instruction decoder very complex. It will also complicate the design of the processor.

3. After finalising the number of instructions, we need to finalise the different types of instructions. If we are designing an ISA for extensive numerical computation, then we should have many arithmetic operations. If we are designing an ISA for processing text, then we should have many instructions that can process strings (pieces of text). In the case of *SimpleRisc* we devoted 6 instructions to arithmetic operations, 3 instructions to shift operations, 3 instructions to logical operations, 3 instructions to data transfer, 5 instructions to branch operations, and designated 1 instruction as no-op (no operation). We chose this distribution because we expect to run a lot of general purpose programs that will have complex arithmetical and logical constructs. We could have very well gotten rid of an instruction such as *mod* and replaced it with a sophisticated branch instruction, if we wanted to look at programs that will have a lot of branches. These subtle tradeoffs need to be evaluated thoroughly.

4. Once, we have finalised the broad types of instructions and the distribution of instructions across these types, we come to the actual instructions themselves. In this case also, we want to make the common case fast. For example, there is no point in having a division

instruction in programs that do not have divisions operations. Secondly, we need to decide the format of each instruction in terms of the number and type of operands. For example, in *SimpleRisc* , all our arithmetic operations are in the 3-address format. If there is a requirement from the side of processor designers that they want to reduce the number of registers, then we can opt for the 2-address format. Alternatively, if we want to process a massive amount of information in one go such as add a list of 10 numbers, then we can even have a 11-address format instruction.

5. Once the format of the instruction is decided, we need to decide on the different addressing modes. This decision has many ramifications. For example, if we allow the register-indirect addressing mode in arithmetic instructions, then we need to add additional hardware to access the memory and fetch the operand values. On the other hand, if we have a register-only addressing mode for arithmetic instructions, then their implementation will be fast. However, the flip side is that we will need more registers, and more dedicated load-store instructions to access memory. This tradeoff needs to be kept in mind.

6. Once we have designed the set of instructions, we need to decide a proper encoding for it. The main aim should be to reduce the work of the instruction decoder. It is best to have a small set of generic instruction formats that the decoder can quickly discern. We need to balance elegance and efficiency such that the decoder can be simple yet efficient.

3.4 Summary and Further Reading

3.4.1 Summary

Summary 3

1. *Assembly language is a textual representation of machine instructions. Each statement in an assembly language program typically corresponds to one machine instruction.*

2. *An assembler is a program that converts an assembly language program to machine code.*

3. *An assembly language is specific to an ISA and an assembler.*

4. *Assembly language is a vital tool for writing efficient programs, and for designing the core routines of operating systems, and device drivers.*

5. *Hardware designers learn assembly languages to understand the semantics of an ISA. It tells them what to build.*

6. *An assembly language program typically assumes a Von Neumann machine augmented with a finite set of registers.*

7. *A typical GNU assembly file contains a list of sections. Two important sections are text and data. The text section contains the assembly statements that correspond to machine code. The data section holds data and constants that the program will need during its operation.*
8. *A typical assembly statement contains an optional label to uniquely identify it, an instruction with a set of operands, and an optional comment. Instead of an instruction, it can also contain a directive that is a command to the assembler.*
9. *There are typically four types of generic assembly instructions:*
   (a) *Data processing instructions – arithmetic and logical*
   (b) *Data transfer instructions – move, load, and store*
   (c) *Branch instructions – branch, function call, return*
   (d) *Exception generating instructions – transfer control to the operating system*

   An assembly language for a specific ISA also contains some machine specific instructions also that are mainly used to set its configuration or invoke some special feature.
10. *The semantics of operands is also known as the addressing mode.*
11. *The main addressing modes are immediate (specify constant in instruction), register-direct (specify the register's name in the instruction), register-indirect (a register contains the memory address), and base-offset (the offset is added to the memory location in the base register).*
12. *We designed the SimpleRisc assembly language that contains 21 instructions. It is a complete RISC ISA.*
13. *We designed an encoding for each SimpleRisc instruction. We broadly defined three instruction formats*

    branch *Contains a 5 bit opcode and 27 bit offset.*

    register *Encodes a 3-address instruction with two register source operands and one register destination operand.*

    immediate *Encodes a 3-address instruction that has an immediate as one of the operands.*

In this chapter we have looked at the generic principles underlying different flavors of assembly language. We constructed a small assembly language of our own for the *SimpleRisc* ISA, and proceeded to encode it. This information is sufficient to design a basic processor for *SimpleRisc* in Chapter 8. However, we would like to strongly advise the reader to at least study one of the chapters on real world assembly languages – either ARM (Chapter 4) or x86 (Chapter 5). Studying a real language in all its glory will help the reader deepen her knowledge, and she can appreciate all the tricks that are required to make an ISA expressive.

3.4.2 Further Reading

Instruction set design and the study of assembly languages are very old fields. Readers should refer to classic computer architecture textbooks by Henessey and Patterson [Henessey and Patterson, 2010], Morris Mano [Mano, 2007], and William Stallings [Stallings, 2010] to get a different perspective. For other simple instruction sets such as *SimpleRisc* , readers can read about the MIPS [Farquhar and Bunce, 2012], and Sparc [Paul, 1993] instruction sets. Their early variants are simple RISC instruction sets with up to 64 instructions, and a very regular structure. Along with the references that we provide, there are a lot of excellently written tutorials and guides on the web for different ISAs.

Since the last 10 years, a trend has started to move towards virtual instruction sets. Programs compiled for these instruction sets need to be compiled once again on a real machine such that the virtual instruction set can be translated to a real instruction set. The reasons for doing so shall be described in later chapters. The Java language uses a virtual instruction set. Details can be found in the book by Meyer et. al. [Downing and Meyer, 1997]. Readers can also refer to a highly cited research paper that proposes the LLVA [Adve et al., 2003] virtual instruction set.

Exercises

Assembly Language Concepts

Ex. 1 — What is the advantage of the register-indirect addressing mode over the memory-direct addressing mode?

Ex. 2 — When is the base-offset addressing mode useful?

Ex. 3 — Consider the base-scaled-offset addressing mode, which directs the hardware to automatically multiply the offset by 4. When is this addressing mode useful?

Ex. 4 — Which addressing modes are preferable in a machine with a large number of registers?

Ex. 5 — Which addressing modes are preferable in a machine with very few registers?

Ex. 6 — Assume that we are constrained to have at the most two operands per instruction. Design a format for arithmetic instructions such as add and multiply in this setting.

Assembly Programming

Ex. 7 — Write simple assembly code snippets in *SimpleRisc* to compute the following:

i) $a + b + c$

ii) $a + b - c/d$

iii) $(a + b) * 3 - c/d$

iv) $a/b - (c * d)/3$

v) $(a \ll 2) - (b \gg 3)$ (($\ll$ (left shift logical), $\gg$ (left shift arithmetic))

Ex. 8 — Write a program to load the value $0xFFEDFC00$ into $r0$. Try to minimise the number of instructions.

Ex. 9 — Write an assembly program to set the 5th bit of register $r0$ to the value of the 3rd bit of $r1$. Keep the rest of the contents of $r0$ the same. The convention is that the LSB is the first bit, and the MSB is the 32nd bit. (Use less than or equal to 5 *SimpleRisc* assembly statements)

Ex. 10 — Write a program in *SimpleRisc* assembly to convert an integer stored in memory from the little endian to the big endian format.

Ex. 11 — Write a program in *SimpleRisc* assembly to compute the factorial of a positive number using an iterative algorithm.

Ex. 12 — Write a program in *SimpleRisc* assembly to find if a number is prime.

Ex. 13 — Write a program in *SimpleRisc* assembly to test if a number is a perfect square.

Ex. 14 — Given a 32-bit integer in $r3$, write a *SimpleRisc* assembly program to count the number of 1 to 0 transitions in it.

* **Ex. 15** — Write a program in *SimpleRisc* assembly to find the smallest number that is a sum of two different pairs of cubes. [Note: 1729 is the Hardy-Ramanujan number. $1729 = 12^3 + 1^3 = 10^3 + 9^3$].

Ex. 16 — Write a *SimpleRisc* assembly program that checks if a 32-bit number is a palindrome. Assume that the input is available in $r3$. The program should set $r4$ to 1 if it is a palindrome, otherwise $r4$ should contain a 0. A palindrome is a number which is the same when read from both sides. For example, 1001 is a 4-bit palindrome.

Ex. 17 — Design a *SimpleRisc* program that examines a 32-bit value stored in $r1$ and counts the number of contiguous sequences of 1s. For example, the value:

01110001000111101100011100011111

contains six sequences of 1s. Write the result in $r2$.

** **Ex. 18** — Write a program in *SimpleRisc* assembly to subtract two 64-bit numbers, where each number is stored in two registers.

** **Ex. 19** — In some cases, we can rotate an integer to the right by n positions (less than or equal to 31) so that we obtain the same number. For example: a 8-bit number 11011011 can

be right rotated by 3 or 6 places to obtain the same number. Write an assembly program to *efficiently* count the number of ways we can rotate a number to the right such that the result is equal to the original number.

** **Ex. 20** — A number is known as a cubic Armstrong number if the sum of the cubes of the decimal digits is equal to the number itself. For example, 153 is a cubic Armstrong number ($153 = 1^3 + 5^3 + 3^3$). You are given a number in register, $r0$, and it is known to be between 1 and 1 million. Can you write a piece of assembly code in *SimpleRisc* to find out if this number is a cubic Armstrong number. Save 1 in $r1$ if it is a cubic Armstrong number; otherwise, save 0.

*** **Ex. 21** — Write a *SimpleRisc* assembly language program to find the greatest common divisor of two binary numbers u and v. Assume the two inputs (positive integers) to be available in $r3$ and $r4$. Store the result in $r5$. [HINT: The gcd of two even numbers u and v is $2 * gcd(u/2, v/2)$]

Instruction Set Encoding

Ex. 22 — Encode the following *SimpleRisc* instructions:

i) *sub sp*, *sp*, 4

ii) *mov* $r4$, $r5$

iii) *addu* $r4$, $r4$, 3

iv) *ret*

v) *ld* $r0$, [*sp*]

vi) *st* $r4$, 8[$r9$]

Design Problems

Ex. 23 — Design an emulator for the *SimpleRisc* ISA. The emulator reads an assembly program line by line, checks each assembly statement for errors, and executes it. Furthermore, define two assembler directives namely *.print*, and *.encode* to print data on the screen. The *.print* directive takes a register or memory location as input. When the emulator encounters the *.print* directive, it prints the value in the register or memory location to the screen. Similarly, when the emulator encounters the *.encode* directive it prints the 32-bit encoding of the instruction on the screen. Additionally, it needs to also execute the instruction.