MIPS Assembly Language Notes for CSE220

Amos Omondi

November 8, 2023

Chapter 1

Introduction

Within a computer, programs (instructions) and the data on which they operate are represented in binary—*machine code*. *Assembly language* form is a more human-readable representation of such machine code and is produced by a compiler (or similar software) or written directly. In particular, machine operations are expressed using mnemonics that indicate what the operations are, and symbolic names are typically used to refer to data storage elements.

The assembly-language programmer's view of a computer is in terms of its *instruction-set architecture* (ISA), which specifies, among other things, the operations that can be performed by the machine and the corresponding operands (and their types, sizes, and where their are located).

This chapter is a short introduction to basic computer organization and the MIPS ISA, assembly language, and related software. One section of the chapter includes a short but non-trivial fragement of assembly-language code that shows several aspects of such code. This example will be used as an extended running example in subsequent chapters.

1.1 Basic computer organization

Figure 1.1 shows the five main components that make up a basic computer:

- a unit for input, e.g., a keyboard,
- a unit for output, e.g., a screen,
- an arithmetic-and-logic unit (ALU),
- a memory (storage) that holds instructions and data, and
- a control unit.

The control unit directs the actions of the other units. The ALU, as its name implies, carries out arithmetic and logic operations. The *central processing unit* (CPU)—or just *processor*—is the combination of the control unit and the ALU (and any directly associated storage elements).

The input unit is used to initially enter instructions and data into the computer and thereafter to enter data as needed by the processor. In practice, there may be more that one input unit, but for our purposes it will suffice to assume just one—the keyboard. The output unit is used to transmit results

from the computer. As with the input case, there may be more than one output unit, but here it suffices to assume just one—the screen.

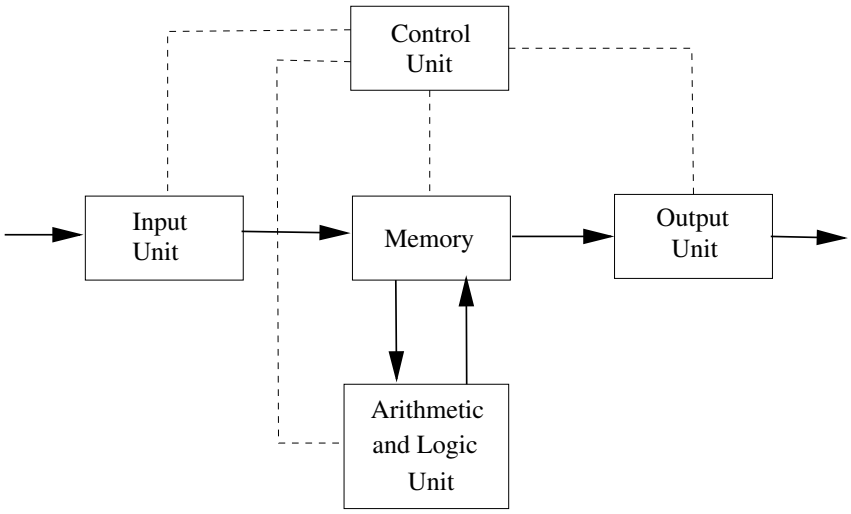

Figure 1.1: Basic computer organization

Under the direction of the control unit, the instructions that make up program code are fetched, one at a time[1] from the memory. The control unit then determines the operations specified within each instruction and directs the other units to carry them out: performing an arithmetic operation, obtaining operand data from the input unit or from the memory, sending results to the output unit or to memory, and so forth.

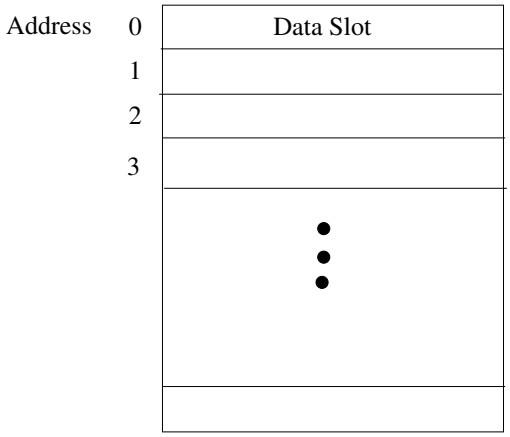

Figure 1.2: Memory

[1]This view is not strictly accurate; in the actual implementation of current processors, multiple instructions may be fetched at a time and processed in an order other than that in which they appear in the code. Nevertheless, it will suffice for our purposes because, the *effect* of any processing must be the same as "one-at-a-time, in-order processing", which is what the programmer assumes.

The main memory (or just "memory") may be viewed as a one-dimensional array of slots that hold data or instructions. Each slot is indexed by an *address*; that is, an address uniquely identifies a single location in memory (Figure 1.2). Getting information (instruction or data) from memory is known as *reading* and requires the specification of the address of the location in memory to be read from. The converse is known as *writing* and requires both the address of the location to write into and the information to be written. *Read* and *write* operations from and to memory are also known as *load* and *store* operations.

The slots of memory hold units of information that are all of the same size. All such information is expressed in bits, but a single bit is typically too small a unit for addressing; addressing at such a level might require an excessive number of bits and can also lead to inefficiencies in processing. Thus the smallest addressable unit is usually a byte (8 bits). Nevertheless, even a byte is often too small an addressable unit for most general-purpose processing. The most convenient unit is a *word*, which in most cases (but not always) is as many bits as are required for an address, or the largest possible representable integer, or a typical machine instruction. The most common word sizes today are 32 bits and 64 bits; and some architectures also allow half of either—a *half-word*—as an addressable unit.

The ordering of bits or of bytes within a larger unit (e.g., a word) may be considered to be from the least significant (right-hand) to the most significant (left-hand) end, or in the other direction—with the numbering of bits starting at 0—and is known as the "endianness": *big endian* order is from left to right, and *little endian* order is from right to left. Figure 1.3 shows 32-bit examples, with the values 13, 43, 5, and 99 stored in the corresponding bytes.

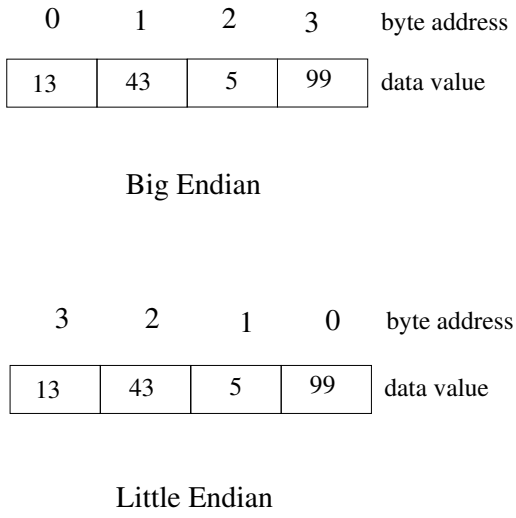

Figure 1.3: Endianness

If in processing an instruction the processor were at all times to fetch operands directly from the memory and write results directly to memory, its effective speed would be determined by that of the memory. But, typically, the hardware circuits used to construct the processor operate at much higher speeds than those used for the memory. So, to ensure that the processor's operational speed is not limited by that of memory, a small amount of very-fast, programmer-addressable intermediate storage—known as *registers*—are included between the memory and the ALU and used to facilitate much faster operation than would be possible with direct access to memory. With judicious coding it can be ensured

that most of the time the operands required for an operation in progress are available in registers, and memory accesses are therefore limited. Corresponding results are initially written to the registers and to memory only when absolutely necessary.

The set of operand registers is known as a *register file*, and from a programmer's perspective it may be viewed similarly to main memory; that is, as a set of data slots, each of which has an associated address (a *register number*), but of a much smaller size (i.e., number of data slots). (See Figure 1.4.)

In addition to the operand registers, there are other registers within the processor. Some of these are directly accessible to the programmer, and some are not. (The latter are said to be "architecturally invisible".) One of the most important of such registers is the *program counter*, which at any given instant "points" to—i.e., contains the memory address of—the next instruction to be processed.

Figure 1.4: Register file

1.2 Introduction to MIPS ISA

The evolution of the MIPS architecture started around 1985, and currently there are three main architectures:

- MIPS32, with 32-bit integers and addresses,
- MIPS64, with 64-bit integers and addresses,
- microMIPS, which are versions of MIPS32 and MIPS64 that are optimized for code density.

And within these groups there are 32-bit MIPS I and II; 64-bit MIPS III, IV, and V; MIPS32 and MIPS64, each in six sequential "releases"; microMIPS in Releases 3 to 6 of each of MIPS32 and MIPS64; and several "modules" in each of the preceding Releases 5 and 6.

For an introduction to assembly-language programming it will suffice to focus on just the MIPS32, since for our purposes the basic principles are the same for all the architectures.

MIPS32 has thirty-two general-purpose registers, each of 32 bits, that the programmer may use for operands and results, addresses, and so forth. We shall refer to these symbolically as \$0, \$1, . . . , \$31; that is, \$*n* means "register *n*".

Two of the registers have special functions: $0 is hardwired to the integer value zero (i.e., always contains that value), and $31 has a special role in function and procecure calls. In principle, the programmer may use any register other than $0 as he or she see fit, but we shall see that there are near-standard conventions of usage that impose some restrictions.

There are also three special-purpose registers in the processor: a 32-bit *Program Counter* that always points to (i.e., contains the address of) the next instruction to be processed and a pair of registers (HI and LO), each 32 bits wide, that are used in combination in Multiply, Divide, and Multiply-Add instructions. The contents of the Program Counter can be modified indirectly by several instructions

Memory in MIPS32 consists of 32-bit words. Bytes within a word may be in big-endian order or little-endian order (Figure 1.3), depending on the actual implementation of the architecture, but, otherwise specified, we shall always assume little-endian order.

There is no inherent meaning associated with an arbitrary string of bits, but relative to a typical high-level language, we may consider datatypes in terms of the units of data that can be specified directly in assemby language instructions: character (8 bits), short integer (16 bits), integer (32 bits), single-precision floating-point (32 bits), and double-precision floating-point (64 bits). Since each register and each memory word is 32 bits wide, a double-precision floating-point representation takes up two words or two registers.

There are instructions that operate on bits, but only bitwise in 32-bit groups—not directly on individual bits. Memory addresses are 32-bit integers.

The smallest addressable unit is a byte; that is, the addresses 0, 1, 2, 3, 4, 5, ... are addresses of bytes. As there are four bytes in a word, word addresses differ by four: 0, 4, 8, 12, 16, ... Addressing may also be of half-words, in which case the addresses differ by two. The size of the unit addressed is specified in the instruction at hand.

A MIPS32 instruction is 32 bits wide and belongs to one of the following five groups.

Load and store instructions

These are the only instructions that access memory; all other instructions operate on operands in registers and return results to registers. Load instructions read from memory, and Store instructions write into memory. There are several instructions of each of the two types, according to the size of the data unit to be read or written, the type of data, and so forth.

Computational instructions

These are arithmetic and logic instructions and include Add, Subtract, Multiply, Divide, Shift and Rotate, And, Or, Xor, comparison instructions, and so forth. Each such instruction operates on two operands; one operand is always in a register, and the other may be in a a register or may be a directly-given constant (an *immediate*) operand. A result always goes into a register. The arithmetic operations here are all integer operations.[2]

Jump and branch instructions

Instructions are normally[3] processed in the sequence in which they appear in a piece of code. A Branch or Jump instruction breaks that sequential flow and transfers control to another instruction at a specified address. Branching may depend on a condition, determined from the contents of two registers, or may be unconditional. Jumps are always unconditional.

[2]Floating-point operations are carried out in a "co-processor" that is attached to the primary MIPS32 processor.

[3]Exceptions, such as certain types of error, can occur during the processing of an instruction.

In assembly language the target of a Branch is specified as a label on the target instruction, and in machine code the target address is encoded as an offset relative to the address in the Program Counter. The target address of a Jump too may be specified similarly, as a label—but with the address specified directly in the corresponding machine code—or as the contents of a register.

Miscellaneous instructions

There are five types of miscellaneous instructions, of which the most significant (for our purposes) are System-Call, Breakpoint, and Trap instructions—together known as *Exception* instructions—that transfer control to a kernel-mode (i.e., operating system) exception handler. The first two types of exception-instructions cause unconditional exceptions, and the other causes conditional exceptions based on the results of comparisons.

Coprocessor instructions

A MIPS processor may have up to four attached co-processors. One of these is a System Controller that handles exceptions and some memory-management functions. Another is a Floating-Point Unit that carries out floating-point arithmetic operations; this unit contains 32 floating-point registers, designated $f0, $f1, $f2, ..., $f31.

1.3 An example MIPS program

The main components of assembly-language code are

- instructions and their operands, expressed in symbolic form;
- user-defined identifiers, which start with an alphabetic character and may include both alphabetic and numeric characters;
- sequences of digits that represent numbers;[4]
- special symbols: $, ., +, -, ‘, ”, (,), and so forth.

Instructions are usually written as one per line.

We now give our first example of assembly-language code, by way of a simple example that can be readily understood, even at this early stge. Suppose we wish to write code for the addition of the first 100 nonzero integers. A straightforward algorithm for the task consists of running a counter from 1 to 100 and adding each value to a running sum that is initially zero:

1. Initialize Sum to zero
2. Initialize Counter to 1
3. Initialize Counter-Limit to 100
4. Add to Sum the value of the Counter
5. Increment the Counter
6. If Counter $\leq$ Counter Limit, then go back to (4)

[4]For simplicity, we shall not always distinguish between a number and its representation (which may be in binary, octal, or hexadecimal), unless such a distinction is necessary to avoid confusion. “Number” may mean either.

Since arithmetic operations must be on operands in registers, the Sum, Integer-Counter, and Counter-Limit values must be held in registers; let us use registers 8, 9, and 10 for those. We may then translate the algorithm into the following assembly-language code

EXAMPLE 1.1: PROGRAM TO ADD FIRST 100 NONZERO INTEGERS

```
        li $8, 0              # sum = 0
        li $9, 1              # counter = 1
        li $10, 100           # counter-limit = 100
again:  add $8, $8, $9        # sum = sum + counter
        addi $9, $9, 1        # counter = counter + 1
        ble $9, $10, again    # loop back, if not done
```

□

The # denotes the start of a comment that is not part of the code. Everything from (and including) the # to the end of the line is discarded during the translation into machine code.

The reader, on the basis of the following explanation, may find it helpful to trace through, with paper and pencil, the execution of the code, with 100 replaced with a suitably smaller number.

The first instruction in the code fragment is a Load-Immediate (li) instruction whose effect is the placement of the given constant ("immediate") operand 0 into register 8. The next two instructions are similar and place 1 and 100 into registers 9 and 10, respectively.

The fouth instruction is an Add whose effect is that the values in last two named registers (8 and 9 here) are added together and the result placed in the first named register (8 here). The instruction here also has an associated user-defined label, "again", that is used later to refer to the instruction. Such a label does not correspond to actual machine code but is translated into an instruction address at the point of reference.

The fifth instruction is an Add-Immediate (addi) whose effect is the addition of the given immediate operand to the value in the second-named register (9 here), with the result going into the first-named register (9 again here).

The last instruction is a branch instruction: ble is an abbreviation for Branch-on-Less-or-Equal. A comparison is made of the two values in the specified registers (9 and 10 here). If those the value in the first-named register less than or equal to the value in the second-named register , then the next instruction to be processed is that with the specified label ("again" here). Otherwise, the processing continues with the next instruction in sequence (of which there is none here). A loop is thus effected here by going back and repeating the fourth, fifth, and sixth instructions.

We shall refer to the registers that contain the operands as *source registers* and those that hold the results as *destination registers*. In this particular code fragment above, one source register is also the destination register in each of two instructions; that need not always be the case.

1.4 Assemblers, linkers, and loaders

There are three main pieces of software between the assembly-language code that the programmer writes and the actual machine code that is eventually fetched from the memory and executed. The three are the *assembler*, the *linker*, and the *loader*. The processes are correspondingly known as *assembly, linking*, and *loading*. Of the three, we shall be concerned mostly with the assembler.

The assembler takes assembly-language code and produces machine code. Pieces of code may be assembled separately, with references from one piece to another, and then run as, essentially, one larger piece of code. A major task of the *linker* is to resolve such references.

Code and data normally reside on secondary storage (e.g., hard disk) and are brought into main memory for processing as necessary. Therefore, the actual addresses for given code and data will generally not be known until they are placed in memory. Moreover, those addresses can also change every time the same code and data are brought into memory. So, a programmer will mostly refer to memory in symbolic terms, not actual addresses; and, correspondingly, the assembler too will not necessarily use actual addresses. The essential tasks of the *loader* are the placement in memory of instrcutions and data and the appropriate adjustment of addresses at the time of placement.

In addition to the primary task of straightforward translation, the assembler facilitates certain conveniences for the programmer. One of these is the inclusion of *pseudo-instructions* in the assembly language. A pseudo-instruction corresponds to one or more machine-code instructions, and its main purpose is to provide a mnemonic that more clearly indicates what the programmer intends and, perhaps, in a more succint form. For example, to copy (move) the contents of one register to another, the assembly-language instruction that corresponds directly to machine code is add. Thus, for example, the the effect of the instruction

```
add $8, $9, $0
```

is to copy the contents of register 9 into register 8, by addding the contents of register 0 (which always contains zero) to the contents of register 9 and then storing the result in register 8. The same effect may be specifed with the pseudo-instruction move:

```
move $8, $9
```

A slighly more complicated example is the Load-Immediate of the code of Example 1.1 (Section 1.3); this too is actually a pseudo-instruction. An immediate operand is so-called because the constant in question is directly embedded in the binary machine-code instruction and thus does not have to be obtained from elsewhere (register or memory). The width of such an instruction allows for only 16-bit constants, but Load-Immediate has a nominal immediate operand that may be up to 32 bits wide. A Load-Immediate is therefore effected in one of two ways. If the immediate operand can be respresented in 16 bits, then the corresponding real instruction is one that involves a 16-bit immediate; otherwise two real instructions are used, each of which moves 16 bits into the named register. Thus in such a case the pseudo-instruction both indicates the programmer intent and is expressively more succint.

Chapter 2

Assembler and simulator

This chapter is an introduction to SPIM, software that consists of an assembler for the translation of assembly-language code into machine code and a simulator of a MIPs processor for the execution of the translated code.

The first section of the chapter is a brief description of the storage name spaces visible in the assembly language; these are those elements of storage that the programmer can directly access, by speciying some form of address. The second section describes how the assembler is used to allocate memory. The third section is on"system calls" for input, output, and so forth. And the last section is a brief explanation on how to use SPIM.

All numbers given in the text are to be taken as decimal, unless otherwise specified (e.g., as binary, or octal, or hexadecimal).

2.1 Registers and memory

The programmer's storage name spaces consist of certain registers and main memory. There are two main sets of registers: 32 "general-purpose" registers for the main MIPS processor and 32 floating-point registers for a separate co-processor; each register is 32 bits wide. There are also several "special-purpose" registers whose uses are described later.

As indicated in Section 1.2, there are conventions for the use of registers; for the most part these are not obligatory, but they have become near-standard. As part of those conventions, registers are usually referred to by symbolic names that are indicative of their use, instead of simply by numbers. (It is, however, acceptable to use just the numbers.) Table 2.1 shows the names of the general-purpose registers in the most widely-used convention for; the usage and explanation of these names are given in subsequent chapters. The floating-point registers are $f0, $f1, $f2, ..., $f31.

The "t" in $t0 to $t9 indicates that they are "temporaries", a term whose precise meaning is explained later. Using these names, the code of Example 1.1 in Section 1.3 (for the addition of the first 100 integers), that is,

```
        li $8, 0              # sum = 0
        li $9, 1              # counter = 1
        li $10, 100           # counter-limit = 100
again:  add $8, $8, $9        # sum = sum + counter
        addi $9, $9, 1        # counter = counter + 1
        ble $9, $10, again    # loop back, if not done
```

may be replaced with the folowing code.

EXAMPLE 2.1: PROGRAM TO ADD FIRST 100 NONZERO INTEGERS

```
        li $t0, 0
        li $t1, 1
        li $t2, 100
again:  add $t0, $t0, $t1
        addi $t1, $t1, 1
        ble $t1, $t2, again
```

□

Register	Name
$0	$zero
$1	$at
$2, $3	$v0, $v1
$4–$7	$a0–$a3
$8–$15	$t0–$t7
$16–$23	$s0–$s7
$24, $25	$t8, $t9
$26, $27	$k0, $k1
$28	$gp
$29	$sp
$30	$fp or $s8
$31	$ra

Table 2.1: Register names

One of the functions of the assembler is to facilitate the use of memory. The programmer will typically use and refer to memory in one of two ways: *static* and *dynamic*. In static allocation the size of memory required is known at the time the code is written (and, therefore, at the time of assembly); the memory will be allocated at assembly time and will be mostly referred to using symbolic names. In may also be necessary to allocate memory at run-time; for example to accomodate growing data structures, such as lists and trees. Such allocation is dynamic.

The most basic division of memory is into parts for code and data. At the very least, this helps ensure safety—that there will be no attempt to process as instructions what are just data and no attempt to modifiy as data what are just instructions, or that such attempts can be detected. Thus for the programmer, the two main partitions of the memory are the *data segment* (with a static part and a dynamic part) and the *text segment* (for program code).

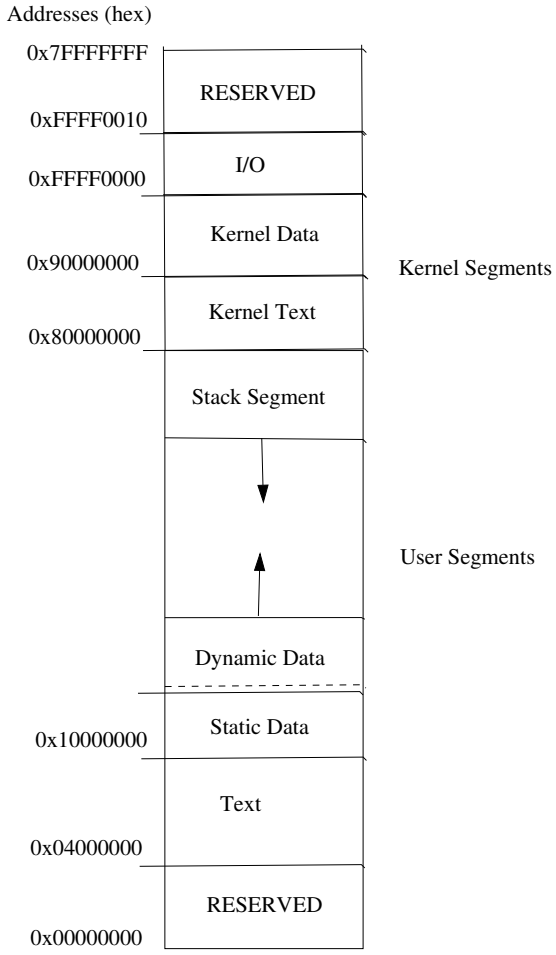

Figure 2.1: MIPS32 memory partitions

"Functions", "procedures", and "subroutines" also require some dynamic memory allocation because they become active and inactive at runtime, and, therefore, memory must be allocated and deallocated on a per-call basis. Since the calls and returns occur in a first-in/last-out order, a stack is the right structure for these. Accordingly, there is a third memory segment for the user: the *stack segment*, in which memory is allocated and freed in a first-in/last-out order.

In addition to the user memory segments, there are also a *kernel data segment*, a *kernel stack segment* , and other parts of memory that are reserved for use in operating system activities. We shall not be much concerned with these.

The complete memory partitioning is shown in Figure 2.1.

2.2 Assembler directives

Assembler directives are "instructions" to the assembler to carry out certain actions. The directives are incuded in the input to the assembler but are not translated into the machine code.

The two most basic directives are `.data` and `.text`. A `.data` directive (of which there may be more than one) indicates to the assembler that what follows should go into the user data segment; and a

`.text` directive works similar for user text segment. All of the following discussions are with respect to the user segments, unless otherwise indicated.

A `.data` directive will usually be followed by one or more directives for the actual allocation of memory in specific units. For example,

```
.word 8, 3, 7
```

directs the assembler to allocate three words in the data segment and initialize their contents to 8, 3, and 7 respectively.

The programmer will usually not know the exact memory locations (i.e., addresses) that are allocated by the assembler and in any case needs a convenient way to refer to them, just as one names variables in a high-level language. For such reference, each allocation may be labelled with a name that is then used as a reference elsewhere in the code. (When the program is loaded into memory to be run, such a label is converted into a memory address at the point of reference.)

Consider the example of some high-level-language code in which the programmer has declared two integer variables, `Sum` and `Count`, with the latter initialized to 10, and two arrays, `ArrayA` and `Vector`, each of five integers, with the former initialized with the values 16, 17, 0, 1, 9. The corresponding declarations here would be

```
.data
        Sum:    .word
        Count:  .word 10
        ArrayA: .word 16, 17, 0, 1, 9
        Vector: .space 20
```

The `.space` sets aside 20 bytes, which, at 4 bytes per integer, is sufficient for 5 integers.

Allocations and deallocations on the stack and the dynamic part of the data segment are done at runtime, under program control, and are discussed later.

The SPIM assembler provides a subset of the MIPS assembler directives. These are given in Table 2.2.

The `.byte` and `.half` directives work in the same way as `.word` but allocate bytes and half-words, respectively, instead of words. The `.single` directive allocates a 32-bit (i.e., single-precision) floating-point representation[1] for each given datum, and the `.double` directive does likewise with a 64-bit (i.e., double precision) representation.

A string is essentially an array of characters, each of which takes up one byte; the internal representation of a character is the corresponding ASCII code. The `.ascii` and `.asciiz` directives each store the given string, but with the latter appending a NULL character at the end.

[1] In the format of the IEEE-754 Standard on Floating-Point Arithmetic.

Directive	Effect
.data *addr*	Items that follow are to be stored in the user data segment. Optional starting address.
.text *addr*	Items that follow are to be stored in the user text segment. Optional starting address.
.byte $b_1, b_2, \ldots, b_n$	Allocate 8 bits for each of the n values given.
.half $h_1, h_2, \ldots, h_n$	Allocate 16 bits for each of the n values given.
.word $w_1, w_2, \ldots, w_n$	Allocate 32 bits for each of the n values given.
.float $f_1, f_2, \ldots, f_n$	Allocate 32 bit bits for each of the n values given. (Floating-point format)
.double $d_1, d_2, \ldots, d_n$	Allocate 64 bits for each of the n values given. (Floating-point format)
.ascii *str*	Allocate memory for the given string.
.asciiz *str*	Allocate memory the given string in memory, with null-termination.
.space *n*	Allocate n bytes of memory.
.align *n*	Align the next datum on a 2^n-byte boundary.
.globl *name*	Declare the given name as global.
.extern *name size*	Declare the given name as global and a label for datum of given size.
.kdata *addr*	Items that follow are to be stored in the kernel dats segment. Optional starting address.
.ktext *addr*	Items that follow are to be stored in the kernel text segment. Optional starting address.
.set at	Enable warnings on use of $at.
.set noat	Disable warnings on use of $at.

Table 2.2: Assembler directives

The `.kdata` and `.ktext` directives are similar to `.data` and `.text` but are for operating-systems-kernel data and text segments.

As indicated in Table 2.1, register 1 is named "at" (for "assembler temporary") because the assembler uses it in actual machine code that corresponds to some pseudo-instructions (in which the registers are not explicitly named). The `.set at` directive enables warnings on subsequent instructions that use $at, and the `.set noat` directive disables such warnings.

The `.globl` directive is used to indicate names (symbols) that may be referenced in separate files, i.e., files other than those in which they appear; the typical use is for pieces of code that are assembled separately. The `.extern` directive is similar, for data of given sizes.

2.3 System calls

The piece of code in Example 2.1 is essentially complete, for its primary function, but it produces no output—i.e., on an output device, such as a screen—which, presumably is what one would want. In most programming systems, basic or standard input and output routines are typically provided—"ready-made" in executable machine code—in a "library" that is accessible to a programmer. That is the norm in assembly-language programming too.

Service	Code	Argument(s)	Result
print integer	1	integer in $a0	
print float	2	float in $f12	
print double	3	double in $f12	
print string*	4	string address in $a0	
print character	11	character in $a0	
read integer	5		integer in $v0
read float	6		float in $f0
read double	7		double in $f0
read string	8	buffer address in $a0, length in $a1	
read character	12		character in $v0
sbrk	9	number of bytes in $a0	address in $v0
exit	10		
exit2	17	result in $a0	
open file	13	file name in $a0 flags in $a1, mode in $a2	file descriptor in $v0
read from file	14	file descriptor in $a0, buffer address in $a1, length in $a2	number of characters read in $v0
write to file	15	file descriptor in $a0, buffer address in $a1, length in $a2	number of characters written in $v0
close file	16	file descriptor in $a0	

*The string must be null-terminated.

Table 2.3: SPIM system services

For input and output, SPIM includes a subset (Table 2.3) of the *system calls* of a standard MIPS

assembly-language programming environment. There is a single, generic system-call instruction (`syscall`) that transfers execution to different routines, according to a "service code" (an integer) that is given as an argument to indicate the service required. The service code is specified in $v0, and any input arguments are specified in registers[2] in one or more of $a0, $a1, $a2, $a3, and $f12. A complete or partial input is returned in $v0 or $f0.

Suppose we wish to extend the code of Example 2.1 so that the computed sum is also printed. The system-call code to print an integer is 1, and the value to be printed should be in $a0.

EXAMPLE 2.2: PROGRAM TO ADD FIRST 100 NONZERO INTEGERS, WITH OUTPUT

```
        li $t0, 0
        li $t1, 1
        li $t2, 100
again:  add $t0, $t0, $t1
        addi $t1, $t1, 1
        ble $t1, $t2, again
        li $v0, 1              # load code for "print integer"
        move $a0, $t0          # place argument (Sum) into $a0
        syscall                # call will print integer in $a0
```

□

The routines in Table 2.2 for printing produce their results on SPIM's console window.

The first three routines for input (reading) get numerical values that are typed into the console window; in each case an entire line is read, but characters after the number given are ignored.

The routine for string input reads a string of the specified length—i.e., a given number of characters—from the console window into into a sequence of memory locations that start at the given address.

`sbrk` allocates a block of memory of the given number of bytes and returns the address of the block in $v0.

`exit` and `exit2` stop program execution. In the latter case the argument in $a0 is the value to be returned by the program.

The last routines are for file manipulation and operate in the same was as the corresponding UNIX library routines. These are discussed later in the text.

Let us now modify the program of Example 2.2 so that instead of adding the first 100 nonzero integers it adds the first *N* nonzero integers, where *N* is provided as input. The system code to read an integer is 5, and the value rread is returned in $v0.

[2] According the convention of the naming in Table 2.1, $a0 to $a3 are used for arguments to functions and procedures, and $v0 and $v1 are result for the results of those. More on that later.

EXAMPLE 2.3: PROGRAM TO ADD FIRST N NONZERO INTEGERS, INPUT AND OUTPUT

```
        li $t0, 0
        li $t1, 1
        li $v0, 5               # load code for "read integer"
        syscall                 # will read integer (N)
        move $t2, $v0           # move into limit register
again:  add $t0, $t0, $t1
        addi $t1, $t1, 1
        ble $t1, $t2, again
        li $v0, 1               # load code for "print integer"
        move $a0, $t0           # prepare argument (Sum)
        syscall                 # will print integer in $a0
```

□

EXAMPLE 2.4: PROGRAM TO ADD FIRST N NONZERO INTEGERS, COMPLETE SPIM CODE

```
    .data                       # start of data segment; empty
                                # here, and may be omitted.
    .text                       # start of text segment
    main:  li $t0, 0
           li $t1, 1
           li $v0, 5            # load code for "read integer"
           syscall              # read integer  N
           move $t2, $v0        # move N+1 into limit register
    again: add $t0, $t0, $t1
           addi $t1, $t1, 1
           ble $t1, $t2, again
           li $v0, 1            # load code for "print integer"
           move $a0, $t0        # prepare argument (Sum)
           syscall              # will print integer in $a0
           li $v0, 10           # load code for "stop"
           syscall              # end execution
```

□

2.4 Using SPIM

To run the program of Example 2.3 on SPIM, it should be complete in two ways: there should certain additional information that the assembler expects, as well as some indication on when the processing of instructions should stop.[3] Without the latter the next word of memory would be taken as an instruction and an attempt made to process it as such. The additional information required includes a `.data`

[3]Strictly, SPIM code treats user code as a subroutine (procedure), whose first instruction is labelled "main", so the proper ending should be a "return from subroutine". Subroutine calls and returns are discussed later, so we use "STOP" for now.

directive, if one is necessary, a `.text` segment for the code, and the label "main" on the first instruction to be processed. And the program should be in a plain-text file whose name ends in ".s".

The data segment for the program here will be empty and thus may be omitted, but the code should be in the text segment. The system service-code to stop execution is 10.

To run the program, open SPIM. It should open with the text segment displayed as shown in Figure 2.2. (If you later want to view the data segment, click on the "Data" to the left of the "Text".) There should also be a console window to enter input and display output. The console window may appear behind the main SPIM window, in which case you may wish to place it where it is more directly visible. If at some point the console window disappears, click on "Window" in the menu at the top, and select "Console".

Next, click on "File" in the menu on top of the screen. Under the drop-menu that appears, select "Load File" and follow the steps to load the program. If you modify your program, then to load it again, click on "File" and then select "Reinitialize and Load File". You should now see your program near the top of the data segment.

To run the program, click on "Simulator" in the menu at the top of the screen and select "Run/Continue". Enter your input (a number for *N*) in the console window. The program should run and display the result in the same window.

If you modify your program, then to load it again, click on "File" and then select "Reinitialize and Load File".

We will later discuss other aspects of the simulator, but they are few, largely strightforward, and you can quickly come to grasp with them just by "playing around" and seeing what happens with the different options.

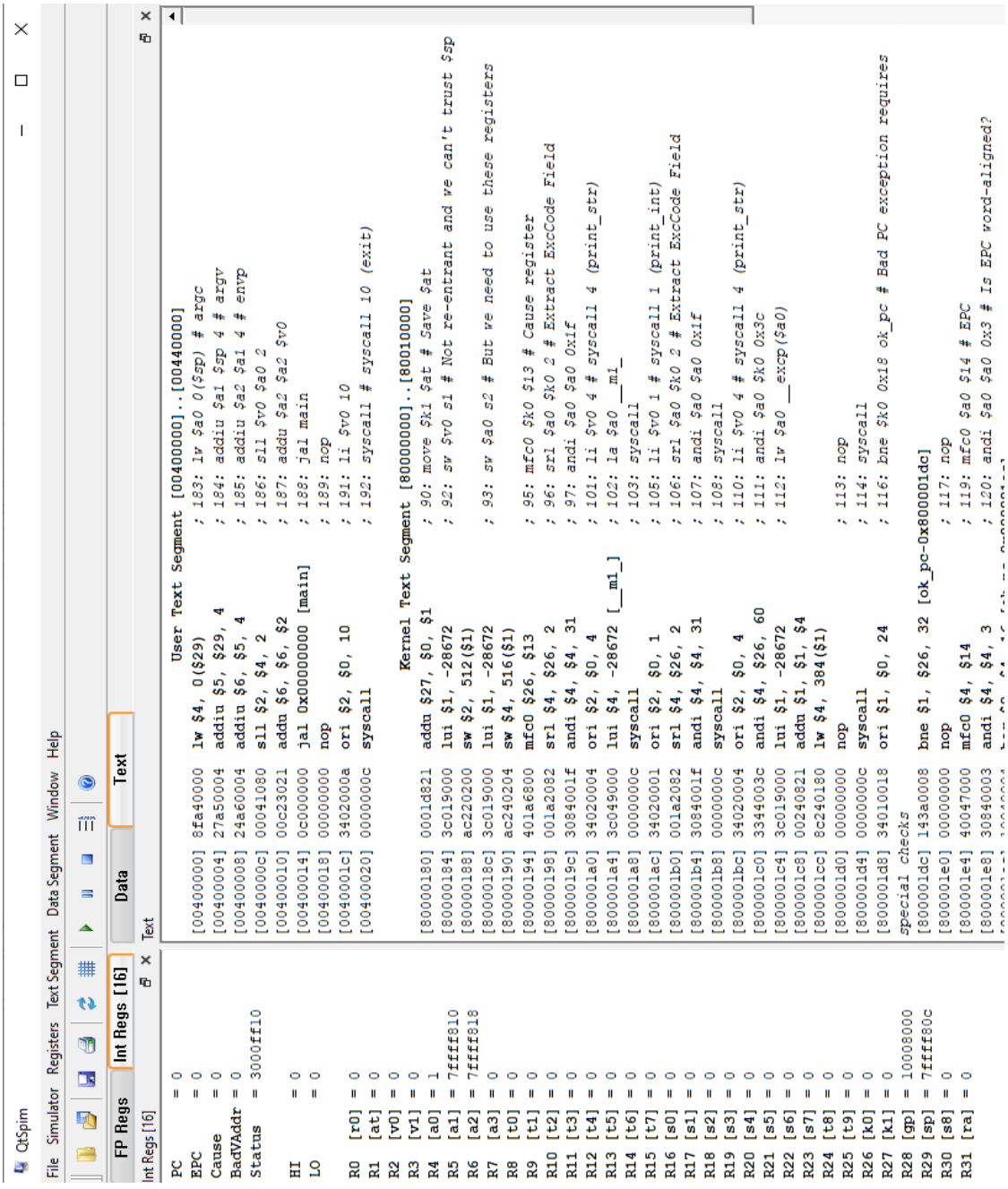

Figure 2.2: SPIM interface