Chapter 2

Instructions: Assembly Language

Reading: The corresponding chapter in the 2nd edition is Chapter 3, in the 3rd edition it is Chapter 2 and Appendix A and in the 4th edition it is Chapter 2 and Appendix B.

2.1 Instructions and Instruction set

The language to command a computer architecture is comprised of **instructions** and the vocabulary of that language is called the **instruction set**. The only way computers can represent information is based on high or low electric signals, i.e., transistors (electric switches) being turned on or off. Being limited to those 2 alternatives, we represent information in computers using **bits** (binary digits), which can have one of two values: 0 or 1. So, instructions will be stored in and read by computers as sequences of bits. This is called machine language. To make sure we don't need to read and write programs using bits, every instruction will also have a "natural language" equivalent, called the assembly language notation. For example, in C, we can use the expression $c = a + b$; or, in assembly language, we can use `add c, a, b` and these instructions will be represented by a sequence of bits $000000 \cdots 010001001$ in the computer. Groups of bits are named as follows:

bit	0 or 1
byte	8 bits
half word	16 bits
word	32 bits
double word	64 bits

Since every bit can only be 0 or 1, with a group of n bits, we can generate 2^n different combinations of bits. For example, we can make 2^8 combinations with one byte (8 bits), 2^{16} with one half word (16 bits), and 2^{32} with one word (32 bits). Please note that we are not making any statements, so far, on what each of these 2^n combinations is actually representing: it could represent a number, a character, an instruction, a sample from a digitized CD-quality audio signal, etc. In this chapter, we will discuss how a sequence of 32 bits can represent a machine instruction. In the next chapter, we will see how a sequence of 32 bits can represent numbers.

2.2 MIPS R2000

The instruction set we will explore in class is the **MIPS R2000 instruction set**, named after a company that designed the widely spread MIPS (Microprocessor without Interlocked Pipeline Stages) architecture and its corresponding instruction set. MIPS R2000 is a 32-bit based instruction set. So, one instruction is represented by 32 bits. In what follows, we will discuss

- Arithmetic instructions
- Data transfer instructions
- Decision making (conditional branching) instructions
- Jump (unconditional branching) instructions

It is important to keep in mind that assembly language is a low-level language, so instructions in assembly language are closely related to their 32-bit representation in machine language. Since we only have 32 bits available to encode every possible assembly instruction, MIPS R2000 instructions have to be simple and follow a rigid structure.

2.2.1 Arithmetic instructions

If we want to instruct a computer to add or subtract the variables b and c and assign their sum to variable a, we would write this as follows in MIPS R2000:

$$\underbrace{\texttt{add}}_{\text{operation}}\ \underbrace{\texttt{a, b, c}}_{\text{operands}} \Longleftrightarrow \texttt{a = b + c;} \text{ as in C language}$$

$$\underbrace{\texttt{sub}}_{\text{operation}}\ \underbrace{\texttt{a, b, c}}_{\text{operands}} \Longleftrightarrow \texttt{a = b - c;} \text{ as in C language}$$

The operation defines which kind of operation or calculation is required from the CPU. The operands (or, arguments) are the objects involved in the operation. Notice that each of the previous MIPS R2000 instructions performs 1 operation and has exactly 3 operands. This will be the general format for many MIPS R2000 instructions since, as we mentioned before, we want MIPS R2000 instructions to have a rigid, simple structure. In the case of `add`, this implies only two operands can be added at a time. To calculate additions of more than 2 numbers, we would need multiple instructions. The following example illustrates this.

Example 2.2.1

The operation `a = b+c+d;` can be implemented using one single instruction in C language. However, if we want to write MIPS assembly code to calculate this sum, we need to write this addition as a series of two simpler additions

```
a = b + c;
a = a + d;
```

such that there are only three operands per operation (addition in this case). The corresponding MIPS code is given by:

```
add a,b,c
add a,a,d
```

So, we need multiple instructions in MIPS R2000 to compute the sum of 3 variables. However, each instruction will be simple (so it can be represented using the 32 bits we have available) and very fast in hardware. Similarly, to compute `a = (b+c)-(d+e);` we proceed as follows

$$\texttt{add } t_0\texttt{,b,c}$$
$$\texttt{add } t_1\texttt{,d,e}$$
$$\texttt{sub a, } t_0\texttt{, } t_1$$

Registers

In a high-level programming language such as C, we can (virtually) declare as many variables as we want. In a low-level programming language such as MIPS R2000, the operands of our operations have to be tied to physical locations where information can be stored. We cannot use locations in the main physical memory for this, as such would delay the CPU significantly (indeed, if the CPU would have to access the main memory for every operand in every instruction, the propagation delay of electric signals on the connection between the CPU and the memory chip would slow things down significantly). Therefore, the MIPS architecture provides for 32 special locations, *built directly into the CPU*, each of them able to store 32 bits of information (1 word), called "**registers**". A small number of registers that can be accessed easily and quickly will allow the CPU to execute instructions very fast. As a consequence, each of the three operands of a MIPS R2000 instruction is restricted to one of the 32 registers.

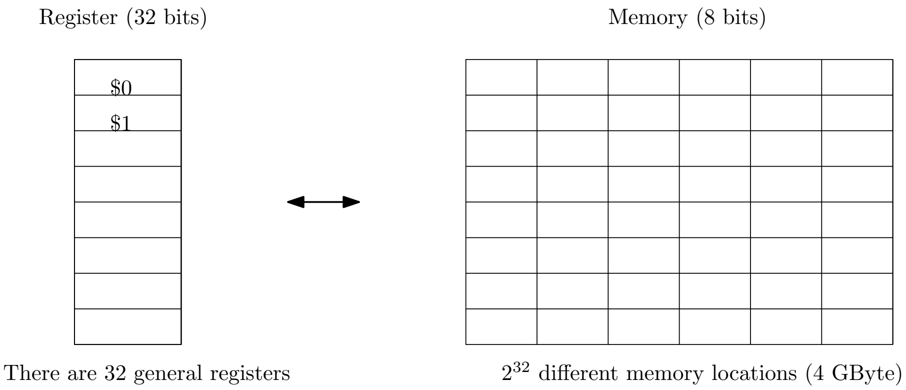
For instance, each of the operands of `add` and `sub` instructions needs to be associated with one of the 32 registers. Each time an `add` or `sub` instruction is executed, the CPU will access the registers specified as operands for the instruction (without accessing the main memory).

The instruction

```
add $1, $2, $3
```

means "add the value stored in the register named `$2` and the value stored in the register named `$3`, and then store the result in the register named `$1`." The notation `$x` refers to the name of a register and, by convention, always starts with a `$` sign. In this text, if we use the name of a register without the `$` sign, we refer to its content (what is stored in the register), for example, `x` refers to the content of `$x`.

Large, complex data structures, such as arrays, won't fit in the 32 registers that are available on the CPU and need to be stored in the main physical memory (implemented on a different chip than the CPU and capable of storing a lot more information). To perform, e.g., arithmetic operations on elements of arrays, elements of the array first need to be *loaded* into the registers. Inversely, the results of the computation might need to be *stored* in memory, where the array resides.


Register (32 bits)
Memory (8 bits)
$0
$1
There are 32 general registers
2^{32} different memory locations (4 GByte)

As shown in the figure above, one register contains 32 bits (1 word) and one memory cell contains 8 bits (1 byte). Thus, it takes 4 memory cells ($4 \cdot 8$ bits) to store the contents of one register (32 bits). For a 32-bit machine (using 32-bit memory addresses), there are 2^{32} different memory addresses, so we could address 2^{32} memory locations, or 4 Gbyte of memory.

To transfer data between registers and memory, MIPS R2000 has *data transfer instructions.*

2.2.2 Data transfer instructions

To transfer a word from memory to a register, we use the **load word** instruction: `lw`

$$\texttt{lw}\quad \underbrace{\texttt{\$r1,}}_{\text{to be loaded}}\ \underbrace{\texttt{100}}_{\text{offset}}\ \underbrace{\texttt{(\$r2)}}_{\text{base register}} \Longleftrightarrow \texttt{r1} = \texttt{mem}[\texttt{100}+\texttt{r2}]$$

Again, this MIPS R2000 instruction performs one operation and has 3 operands. The first operand refers to the register the memory content will be loaded into. The register specified by the third operand, the *base register*, contains a memory address. The actual memory address the CPU accesses is computed as the sum of “the 32-bit word stored in the base register (`$r2` in this case)” and “the offset” (100 in this case). Overall, the above instruction will make the CPU load the value stored at memory address `[100+r2]` into register `$r1`.

Also, it needs to be pointed out that a `lw` instruction will not only load `mem[100 + r2]` into a register, but also the content of the 3 subsequent memory cells, at once. The 4 bytes from the 4 memory cells will fit nicely in a register that is one word long.

To transfer the content of a register to memory, we use the **store word** instruction: `sw`

$$\texttt{sw}\quad \underbrace{\texttt{\$r1,}}_{\text{to be stored}}\ \underbrace{\texttt{100}}_{\text{offset}}\ \underbrace{\texttt{(\$r2)}}_{\text{base register}} \Longleftrightarrow \texttt{mem}[\texttt{100}+\texttt{r2}] = \texttt{r1}$$

The structure of this instruction is similar to `lw`. It stores the content of `$r1` in memory, starting at the memory address obtained as the sum of the offset and the 32-bit word stored in the base register. The 3 subsequent memory cells are also written into (to store all 32 bits stored in `$r1`).

Example 2.2.2

Let `A` be an array of integers (each represented by a 32-bit word), with the base address of `A` stored in register `$3`. Assume that the constant `h` is stored in register `$2`. We can implement

```
A[12] = h+A[8];
```

in MIPS R2000 as

```
lw $0, 32($3)    # load A[8] to $0
add $0, $0, $2   # add h and $0
sw $0, 48($3)    # store the sum in A[12]
```

In the first instruction, we use 32 as the offset since one integer is represented by 4 bytes, i.e., 4 memory cells, so the 8th element of the array is stored 32 bytes away from the base address. Similarly, the last instruction uses an offset of 48 (12 times 4 bytes). The `#` sign allows to insert comments, similar to using "//" in C.

Loading and storing just one byte

To load just one byte from memory (stored, e.g., at memory address `r2 + 100`) into a register, e.g., `$r1`, we can use the following instruction

```
lb $r1, 100 ($r2)
```

This instruction (**load byte**) loads the byte into the 8 rightmost bits of the register (as we will see later, the 8 bits will be sign-extended to 32 bits, to fill the entire register). Similarly, the following instruction (**store byte**) allows to store the 8 rightmost bits of a register, e.g., `$r1`, into memory, at the address `r2 + 100`:

```
sb $r1, 100 ($r2)
```

2.2.3 Adding a constant

The `add` instruction we introduced earlier adds the contents of two registers. To add a constant to the content of a register would require to first load the constant value from memory into some register and then execute an `add` instruction to add the content of two registers. This requires two instructions, including one data transfer between memory and a register, which can be time consuming. Using the **add immediate** instruction, this can be done more efficiently:

`addi $r1, $r2, 4` $\Longleftrightarrow$ `r1 = r2 + 4`

This allows to add a constant and a register value with just one instruction (without data transfer). In general, **immediate instructions** will have two registers and one constant as operands.

2.2.4 Representing instructions in machine language

After writing a program in assembly language, each instruction needs to be translated into a string of 32 bits, i.e., machine language. For example, the assembly instruction `add $8, $17, $18` is translated into machine language as follows:

$$\begin{array}{c}\texttt{add \$8,\$17, \$18}\\ \Updownarrow \\ \underbrace{000000}_{\text{OP}}\underbrace{10001}_{\text{RS}}\underbrace{10010}_{\text{RT}}\underbrace{01000}_{\text{RD}}\underbrace{00000}_{\text{SHAMT}}\underbrace{100000}_{\text{FUNCT}}\end{array}$$

To understand how this translation is performed, we need to look at how the CPU will segment each 32-bit instruction into different **fields**, to understand what operation is expected before executing it. The *instruction format* rigorously defines this segmentation (as indicated above). The meaning of each field is described below.

R-type instruction format

FIELD	**Meaning**	**# of bits**
OP (opcode)	Basic operation	6
RS	1st source register (operand)	5
RT	2nd source register (operand)	5
RD	Destination register (operand)	5
SHAMT	Shift amount	5
FUNCT	Specific functionality	6

The register corresponding to each operand is encoded using 5 bits, which is exactly what we need to represent one of $32 = 2^5$ registers. The SHAMT field will be discussed later, when shift instructions are introduced. The FUNCT field can be used to select a specific variant of the operation specified in the OP field (e.g., OP can specify a shift instruction and FUNCT then indicates whether it is a right-shift or a left-shift)

Using the previous instruction format would only provide 5 bits to represent the third operand, a constant, in an `addi` or `lw` instruction. Therefore, an instruction like, e.g., `addi $r1, $r2, 256`, where the constant value of 256 cannot be represented using only 5 bits (since $11111_2 = 31$), could not be represented in machine language. Similarly, the offset in an `lw` instruction would be limited.

To solve this problem without having to increase the length of instructions beyond 32 bits, MIPS provides for a small number of different instruction formats. The instruction format we described before is called the *R(register)-type* instruction format (where all 3 operands are registers). Instructions like `lw` and `addi` will be represented using the *I(immediate)-type* instruction format (where one operand is a 16-bit constant, allowing much larger constants and offsets).

I-type instruction format

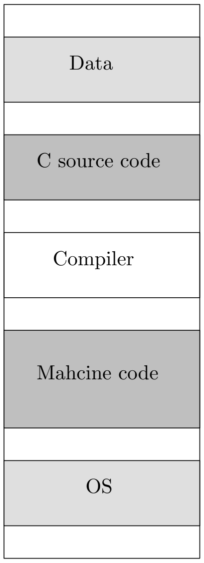
FIELD	**Meaning**	**# of bits**
OP (opcode)	Basic operation	6
RS	Source register (operand)	5
RT	Destination register (operand)	5
Constant	Constant (operand)	16

Note that even though we are using the field name RT, the meaning of RT in an I-type instruction is "destination register", while, for R-type instructions, it refers to a "source register". After analyzing the opcode, the CPU will be able to determine whether the remaining part of the instruction follows the R-type or the I-type format. Immediate instructions and data transfer instructions are using the I-type instruction format. For example,

$$\begin{gathered}\texttt{lw \$8, 2000(\$9)}\\ \Updownarrow \\ \underbrace{100011}_{\texttt{lw}}\,\underbrace{01001}_{\texttt{\$9}}\,\underbrace{01000}_{\texttt{\$8}}\,\underbrace{0000011111010000}_{\texttt{2000}}\end{gathered}$$

2.2.5 Stored-program concept

Essentially, each program is an array of instructions, where each instruction is represented as a 32-bit word. Just like data (integer numbers, characters, etc. represented by a sequence of bits), instructions can be stored in memory. This *stored-program concept* was introduced by von Neumann and architectures that store both data and programs in memory are called **von Neumann architectures**. In a **Harvard architecture**, programs and data are stored in separate memory chips (e.g., DSPs). In a **non-Harvard architecture**, programs and data are stored on one and the same memory chip, which gives rise to the picture below: in one and the same memory, data, C source code, a C compiler program, compiled machine code, the OS program, etc. are all simultaneously present, in many different sections of the memory, as depicted below.



In memory, various types of information are stored.

2.2.6 Decision making (conditional branching) instructions

Conditional branching instructions allow to decide between 2 alternatives, based on the results of a test.

$$\underbrace{\texttt{beq}}_{\text{branch if equal}}\ \underbrace{\texttt{\$r1, \$r2,}}_{\text{compared regs}}\ \underbrace{\texttt{label}}_{\text{branch address}}$$

$\Longleftrightarrow$if `r1==r2`, jump to instruction at `label`
else, go to next instruction

$$\underbrace{\texttt{bne}}_{\text{branch if not equal}}\ \underbrace{\texttt{\$r1, \$r2,}}_{\text{compared regs}}\ \underbrace{\texttt{label}}_{\text{branch address}}$$

$\Longleftrightarrow$if $\texttt{r1}\neq\texttt{r2}$, jump to instruction at `label`
else, go to next instruction

Decision making instructions compare the first two operands and behave differently based on the result of the comparison. In the case of `beq`, the contents of the two registers, `$r1` and `$r2` in this case, will be compared and if they are equal, the CPU will continue with the instruction at the memory address specified by the third operand, `label`. If the compared values are not equal, the CPU will continue with the next instruction. In machine language, the third operand will describe the actual memory address (in bits). In assembly language, the programmer is being relieved from the burden of having to specify the address explicitly and, instead, a symbolic reference, `label`, to the actual address is used, called a "**label**" (this also addresses the problem that the actual memory address of an instruction is only known exactly when the OS allocates physical space in memory to a program). In the case of `bne`, if the compared values are not equal, the CPU will continue execution at `label`, otherwise, it will continue with the next instruction. Let's look at an example.

Example 2.2.3

```
       beq $r1, $r2, label        # program arrives at branch if equal instruction
       sub $r3, $r2, $r5          # program will execute this only if r1 != r2
label: addi $r4, $r2, 3           # program will always execute this instruction
```

Using decision making instructions, we can implement `if` statements, `for` loops, etc., as illustrated in the next example.

Example 2.2.4

Let `A` be an array of 100 integers. The base address of `A`, `0x00000000`, is stored in `$1`. The values of `$4` and `$2` are 400 and 65536, respectively. The C language code

```
for(i=0; i<100; i++)
 A[i] = 65536 +A[i];
```

can be implemented in MIPS R2000 as

```
  loop: lw $3, 0($1)     # load A[i]
        add $3, $2, $3   # A[i] = A[i] + 65536
        sw $3, 0($1)     # store A[i]
        addi $1, $1, 4   # i = i + 1
        bne $1, $4, loop # if i != 100, continue at "loop", o.w. at next instruction
```

`bne` does not check if `i<100` in this case. It is just checking whether `i != 100` or not.

To check whether one register's value is smaller than another register's value, we can use the **set-on-less-than** instruction, which also has a **set-on-less-than-immediate** counterpart.

`slt $r0, $r3, $r4`

⟺if `r3` < `r4`, `r0` is set to 1

else `r0` is set to 0

`slti $r0, $r3, 10`

⟺if `r3` < 10, `r0` is set to 1

else `r0` is set to 0

`slt` and `slti` are similar to `beq` or `bne`, however, there are two differences. First, they test whether one value is smaller than another value and, second, they don't branch to some address, but, instead, set a *flag*, stored in the first operand.

2.2.7 Jump (unconditional branch) instructions

Unconditional branch instructions force the CPU to continue execution with an instruction at a memory address that could be far away from the current instruction, without testing any condition. So, the program "**jumps**" from one line to a (potentially very) different line.

`j label` ⟺ jump to `label`

`jr $ra` ⟺ jump to `ra`

While `j` (**jump**) makes the program jump to the instruction stored at `label`, `jr` (**jump register**) makes the program jump to the instruction stored at the 32-bit memory address that is stored in the register `$ra`. `jr` is represented using the R-type instruction format. To translate a `j` instruction into machine language, the J-type instruction format is used, defined as follows:

J-type instruction format

FIELD	**Meaning**	**# of bits**
OP (opcode)	Basic operation	6
Address	Address	26

Although a memory address is 32 bits long, the length of the address field is only 26 bits in the J-type instruction format. Even more, for conditional branch instructions, translated according to the I-type instruction format, there are only 16 bits available to encode the branching address. We will see later how this is resolved using *absolute* respectively *relative addressing*.

Example 2.2.5

We would like to translate the following C code into assembly language:

```
if (i==j) f=g+h;
else f=g-h;
```

Let's assume that the variables are stored in the following registers:

Register	Content
`$0`	`f`
`$1`	`g`
`$2`	`h`
`$3`	`i`
`$4`	`j`

A possible translation into assembly code goes as follows:

```
           bne $3, $4, subtract # if i!=j then go to "subtract"
           add $0, $1, $2       # f = g + h (skipped if i != j)
           j end                # continue with "end"
subtract:  sub $0, $1, $2       # f = g - h (skipped if i == j)
end:
```

2.2.8 Logical Operations

The following are MIPS instructions for some logical operations.

`and  $1, $2, $3`	bitwise `and` applied to `$2` and `$3`, result stored in `$1`
`or   $1, $2, $3`	bitwise `or` applied to `$2` and `$3`, result stored in `$1`
`andi $1, $2, const`	bitwise `and` applied to `$2` and the 16-bit immediate value `const` (16 most significant bits set to zero), result stored in `$1`
`ori  $1, $2, const`	bitwise `or` applied to `$2` and the 16-bit immediate value `const` (16 most significant bits set to zero), result stored in `$1`
`sll  $1, $2, const`	`shift left logical` will shift the bits in `$2` `const` positions to the left, fill emptied bits with zeroes and store result in `$1`
`srl  $1, $2, const`	`shift right logical` will shift the bits in `$2` `const` positions to the right, fill emptied bits with zeroes and store result in `$1`

The instructions `sll` and `srl` are R-type instructions, where the shift amount is represented in the `SHAMT` field. The field `RS` is set to zero.

Example 2.2.6

The instruction `sll $10, $16, 8` is translated in machine language as follows.

$$\begin{array}{c}\texttt{sll \$10, \$16, \$18}\\ \Updownarrow \\ \underbrace{000000}_{\text{OP}}\,\underbrace{00000}_{\text{RS}}\,\underbrace{10000}_{\text{RT}}\,\underbrace{01010}_{\text{RD}}\,\underbrace{01000}_{\text{SHAMT}}\,\underbrace{000000}_{\text{FUNCT}}\end{array}$$

2.3 Summary of the Basic Instructions

The table below summarizes the different instruction formats and lists some of the instructions we've seen so far as examples, for each format.

Type	6bits	5bits	5bits	5bits	5bits	6bits	instructions
R	OP	RS	RT	RD	SHAMT	FUNC	`add, sub, jr, slt`

I	OP	RS	RT	const/addr			`lw, sw, slti,`
							`beq, bne, lb, sb`

J	OP	address					`j`

Certain related instructions get represented in machine language with the same OP field (6-bit) value, but a different FUNC field (6-bit) value. So, the CPU will only be able to distinguish between them based on the FUNC field. For example, `add` and `sub` instructions are encoded with the same OP field value (`000000`) but different FUNC field values. `add` and `sub` use the same CPU resource (called the ALU). For `add`, the ALU will be instructed to add, while for `sub`, the ALU will be instructed to subtract, all based on the FUNC field value.

2.4 Procedures in Assembly

2.4.1 Procedure Basics

A **procedure** is subroutine, consisting of several lines of code, stored in memory that **performs a specific task** based on the **parameters** that it has been provided with. It might or might not return a result. By defining and using procedures, we

- make programs *easier to understand*
- make programs more *structured*
- allow *code reuse*
- allow programmers to focus on *implementing one specific subtask* of a larger programming project

For example, to file a tax return, one could learn how to go through a series of potentially complex steps and do the paperwork oneself, which could be time-consuming. The other option is to provide a CPA, who is specialized in preparing tax returns, with the important parameters (W-2 numbers, 1099 numbers, etc.), and leave it up to her/him to provide the final tax returns and compute the tax owed or overpaid, without worrying about any of the details of the process. In this example, the CPA can be compared to a specialized procedure, which one provides with the relevant parameters, then does the job and returns the results to whoever called for the specialized service. We don't care about how the procedure does the job: the only interaction with the procedure is by passing parameters to it and receiving results.

In assembly code, we need to go through the following six steps to make use of a procedure:

1. Store the parameters (that need to be passed on to the procedure) where the procedure can access and retrieve them easily and quickly

2. Transfer control to the procedure

3. Have the procedure acquire the storage resources it requires

4. Have the procedure execute its specific task

5. Store the results (that need to be passed back to the calling program) where the calling program can access and retrieve them easily and quickly

6. Return control to the calling program, at the instruction that follows after the procedure call

For steps 1 and 5, we need storage space that is easy and quick to access. Therefore, we'll put the parameters and results of a procedure in **registers**, specifically dedicated for that. There are 4 **argument registers** to store parameters and 2 **value registers** to store results:

- `$a0, $a1, $a2, $a3` : 4 argument registers

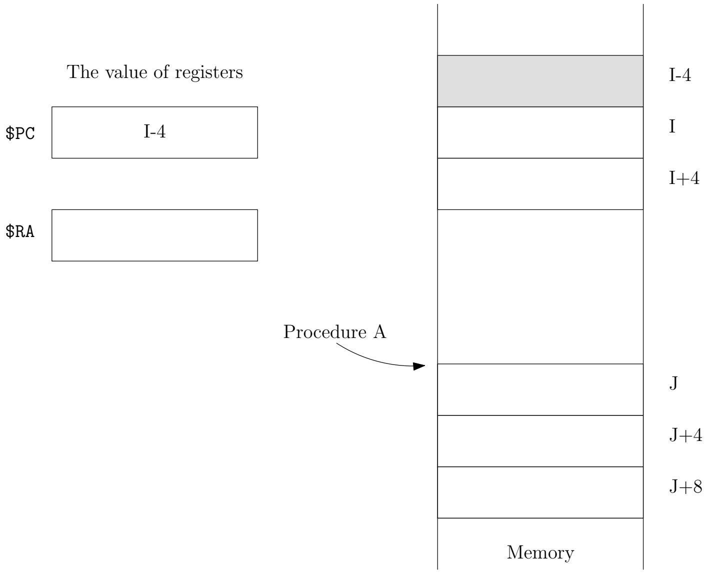
- `$v0, $v1` : 2 value registers

To accomplish steps 2 and 6, we need a register that is dedicated to holding the memory address of the instruction that is currently being executed (remember that every program is stored in memory). We call this the **program counter register**, **$PC** in MIPS. Moreover, to remember which instruction (in the program that calls the procedure) to return to and continue with, after the procedure completes its task, we dedicate one register to store the address of the instruction following the instruction that calls the procedure: the **return address register**, **$RA**. Indeed, the procedure could be called from many different places in our program.

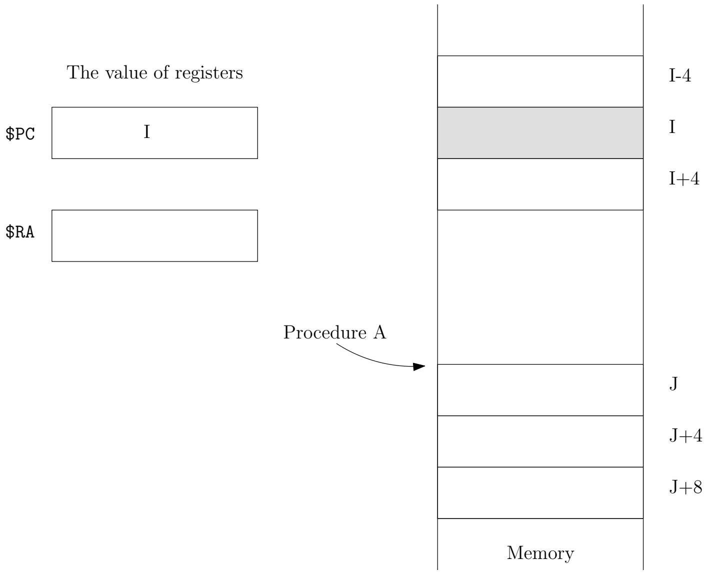
Example 2.4.1

Assume that the current value of `$PC` is I-4. After executing the instruction at memory address I-4, `$PC` is incremented with 4, to I, to execute the next instruction. Assume that the instruction at memory address I tells the CPU to jump to procedure A which starts at the address J and finishes at J+8 as depicted below. Let's take a look at how the value of `$PC` and `$RA` should change throughout this process.

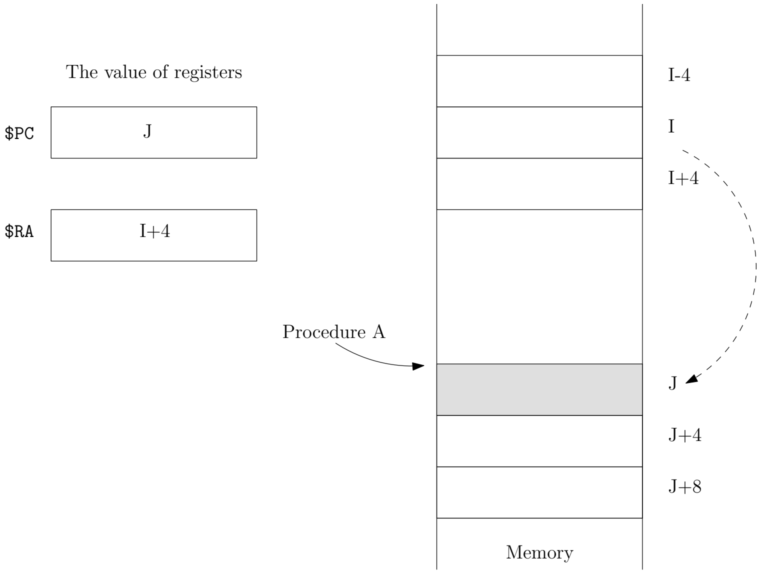
1) Executing the instruction at I-4.



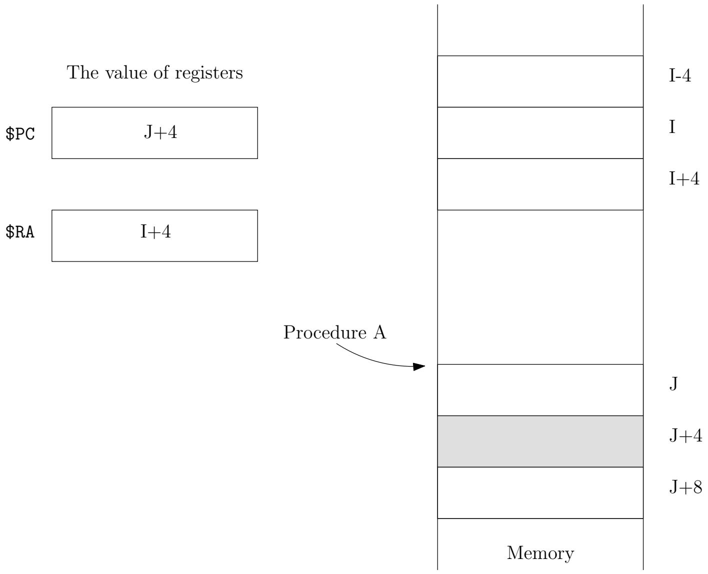
2) The instruction at I wants the CPU to jump to procedure A, and execute procedure A. However, before jumping away, the address of the next instruction, to be executed upon return from A, i.e., I+4, gets stored in `$RA`.



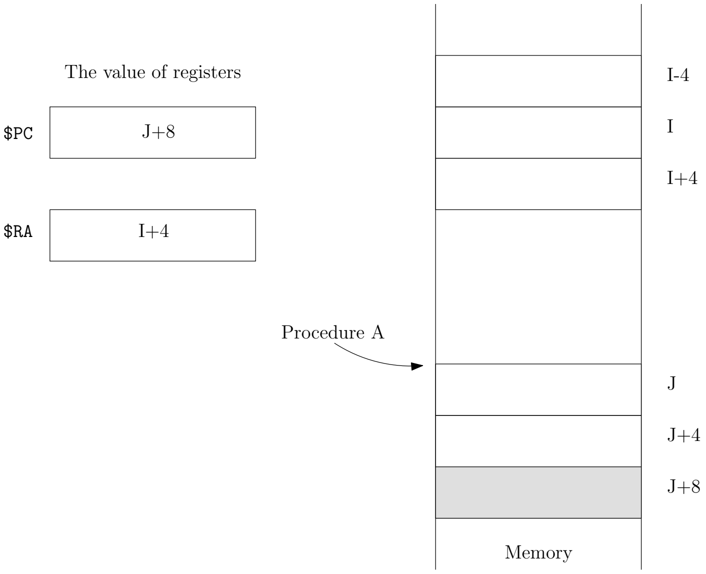
3) After storing the value I+4 in `$RA`, the CPU "jumps" to procedure A by loading the address J into `$PC`.



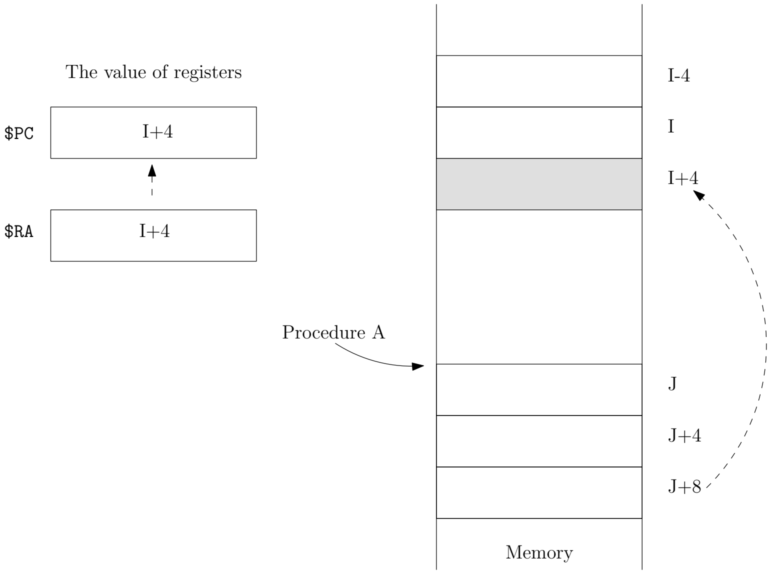
4) Executing the task specific to procedure A, which, in this case, consists of only 1 instruction.



5) The last instruction of procedure A needs to ensure program execution returns to where the procedure was called from. This can be achieved by loading the return address stored in `$RA` into `$PC`.



6) Return to where A was called from and execute the next instruction, after loading the return address stored in `$RA` into `$PC`.



So, in summary,

- To call a procedure at memory address `J`, we need to:

  ```
  RA = PC + 4
  PC = J
  ```

  This can be achieved using a **jump-and-link** instruction, `jal J` ("jumping" to J, after "linking", i.e., saving the return address).

- To return from the procedure, we need to:

  ```
  PC = RA
  ```

  This can be achieved by `jr $RA`.

Note on branching instructions

Now that we have seen how the program counter (a 32-bit register) is used to determine which instruction will be executed next, let's take a closer look at how branching instructions are exactly implemented.

- **Relative addressing for conditional branching instructions**.

  Conditional branching instructions are I-type instructions. That instruction type provides 16 bits to encode the branch address. Since this is too few to encode an actual, full memory address, MIPS R2000 will encode the **word** address of the instruction to branch to (if the branch condition is satisfied), **relative** to the instruction that follows right after the branch instruction. Indeed, since instructions are 1 word long, we can specify the branch address as a number of words before or after the instruction following the branch instruction. So, if the branch is taken, we have:

  ```
  PC = PC + 4 + (4 * 16-bit relative branch address in words)
  ```

  This allows to jump up to 2^{15} words away (forward or backward) from the branching instruction. Since most conditional branching instructions branch to a nearby instruction (as they implement `if` statements, `for` loops, etc.), this is usually enough.

- **Absolute addressing for unconditional branching instructions**.

  The instructions `j` and `jal` are J-type instructions. That instruction type provides 26 bits to encode the target address for the jump. This allows to encode the actual, absolute memory address for the jump, by using the 26 bits as an absolute word address (since instructions are always word-aligned). More specifically, the full 32-bit target address for the jump is formed by concatenating the high order four bits of $PC (these will be the same for all instructions, since instructions are kept in a specific area in memory), the 26 bits specified in the instruction field, and two 0 bits (since instructions are word-aligned):

```
PC = [PC(31:28) 4 * 26-bit jump address in words]
```

This allows j and jal to jump to instructions at memory addresses that are much further away than conditional branching instructions would allow.

Going back to the 6 steps that are required to make use of procedures, we still need to address step 3 (acquiring extra storage space, i.e., registers for the procedure), as well as what to do if there are more than 4 arguments to be passed on to the procedure (more than what fits in the 4 available registers) or more than 2 values to be returned (more than what fits in the 2 available registers). Extra storage space can be acquired by the procedure by moving the content of some of the registers (which the calling program was using) temporarily to the memory. Extra arguments or return values can also be accommodated using memory. The section of memory that is used for all of this, is called the **stack**.

2.4.2 The stack

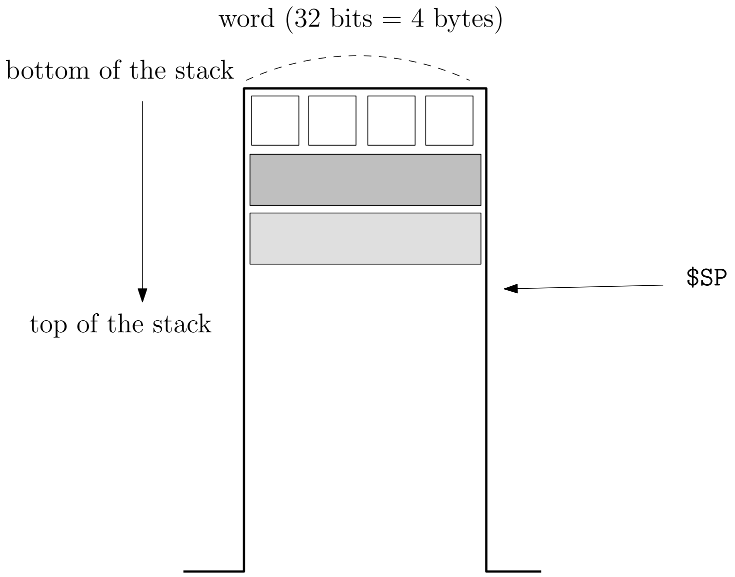
The stack is a data structure that is organized as a last-in-first-out queue: what was added last to the stack, i.e., **pushed** last onto the stack, will be the first item to be removed from it, i.e., **popped** off the stack. The stack is implemented using physical memory and located in the upper end of it. By historical precedent, the stack grows from the higher to the lower memory addresses. One register is dedicated to keeping track of the next available memory address for the stack, to know where to place new data onto the stack (for a push) or find the most recently added item (for a pop): the **stack pointer register**, $SP. One stack entry consists of one word, so $SP gets adjusted by 1 word (4 bytes) for each push or pop operation. Keeping in mind that the stack grows from the higher to the lower memory addresses, $SP gets adjusted as follows:

$$\begin{aligned} \text{PUSH} &: \texttt{SP = SP - 4} \\ \text{POP} &: \texttt{SP = SP + 4} \end{aligned}$$

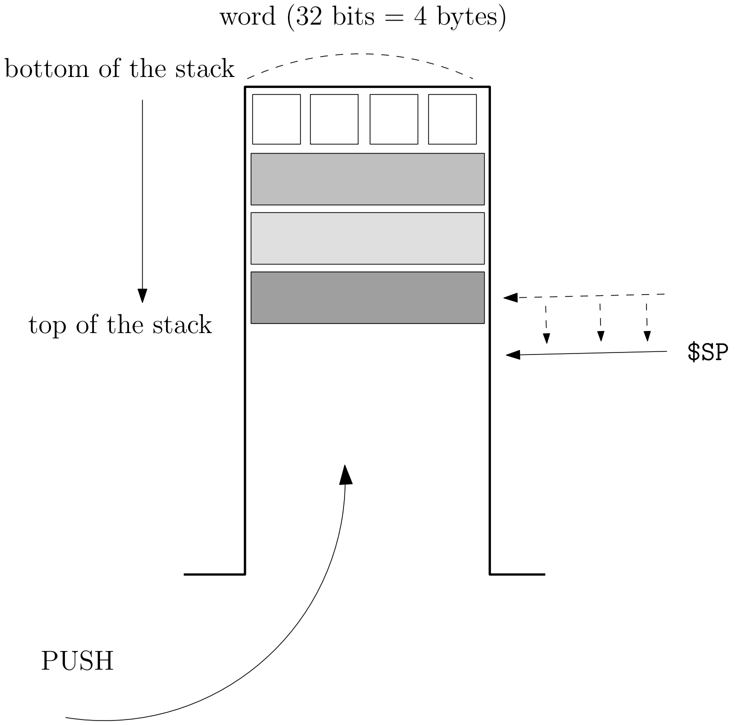
Push and pop operations are implemented using sw respectively lw instructions.

The figures below illustrate the function of the stack pointer $SP and how it is being adjusted when pushing or popping.

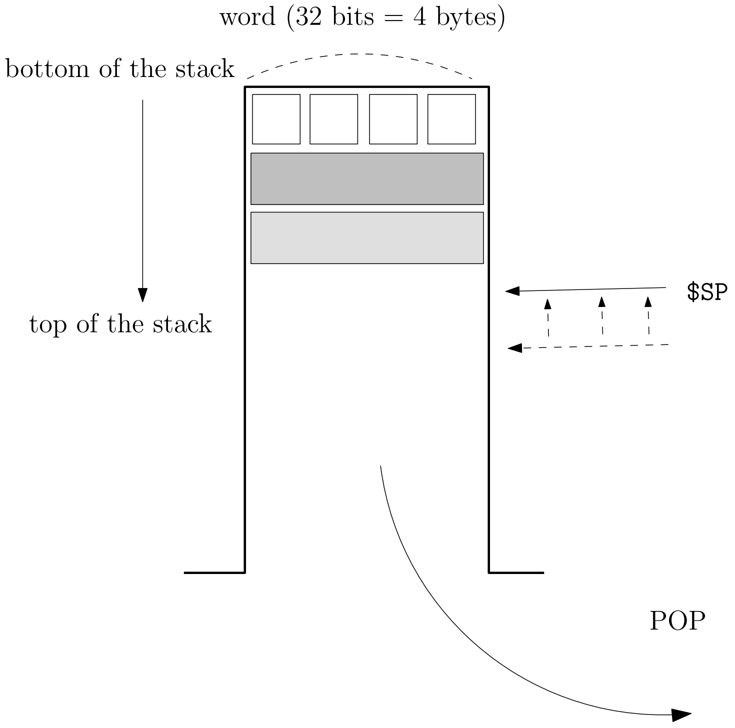
1) The stack, storing words (4 bytes). Note that the bottom of the stack is the top of the memory.



2) Pushing an item onto the stack.



3) Popping an item off the stack.



Using the stack, we can *free up* some of the registers for a procedure by *pushing* their contents onto the stack before starting to execute the procedure. The procedure is then free to use those registers. When the procedure returns the old value of those registers is *restored* by *popping* their old values off the stack. If we need more than 4 arguments, or more than 2 return values, the stack will store the extra arguments and return values as well.

2.4.3 Nested procedures

If a program is written such that procedures call procedures that, in turn, call other procedures, we say it consists of **nested procedures**. For example, below is a high-level picture of a program that executes the main procedure A, which calls a procedure B, which, in turn, calls a procedure C.

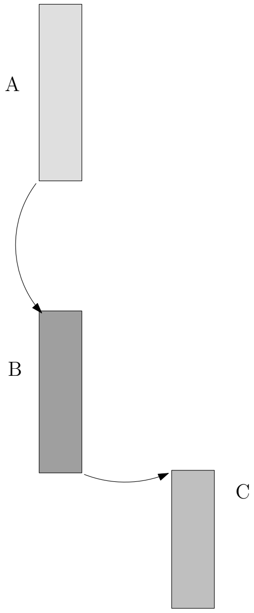


Figure 2.1: Procedure A calls procedure B, which, in turn, calls procedure C.

When A calls B, the arguments A passes on to B are stored in `$a0, ..., $a3` and the return address is stored in `$RA`. When B calls C, the arguments that B passes on to C are stored in `$a0, ..., $a3` and the return address is stored in `$RA` again. Hence, `$RA` and `$a0, .., $a3` are overwritten. This will make it impossible for B to return to A, after C returns to B. Moreover, if B needs some of the arguments it received from A, after C returns to B, those values will no longer be available.

Thus, for nested procedures, the programmer needs to ensure that the values of some of the special registers are preserved. This can be accomplished by pushing the register values onto the **stack**. This is illustrated in the next example.

Example 2.4.2

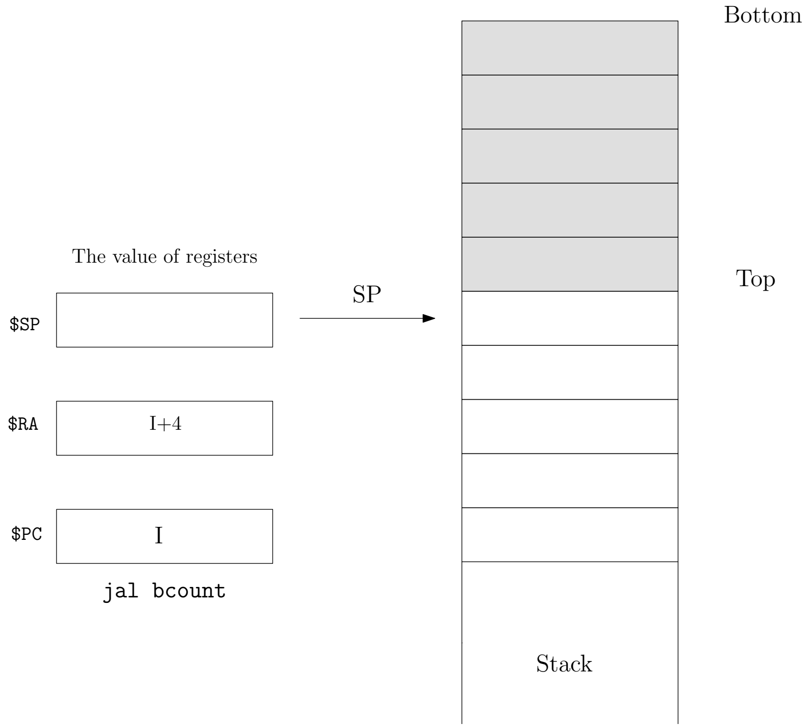
In this example, we focus solely on preserving special register `$RA` through nested procedure calls, assuming that other special registers such as `$a0, ..., $a3` and `$v0`, `$v1` are not used. The MIPS code is given by:

```
I:       jal bcount           # call procedure bcount
I+4:     ...
         ...


bcount : addi $SP, $SP, -4    # make space on stack
         sw   $RA, 4($SP)     # push return address on stack
         ...
         ...
J:       jal  bfind           # call procedure bfind
J+4:     ...
```

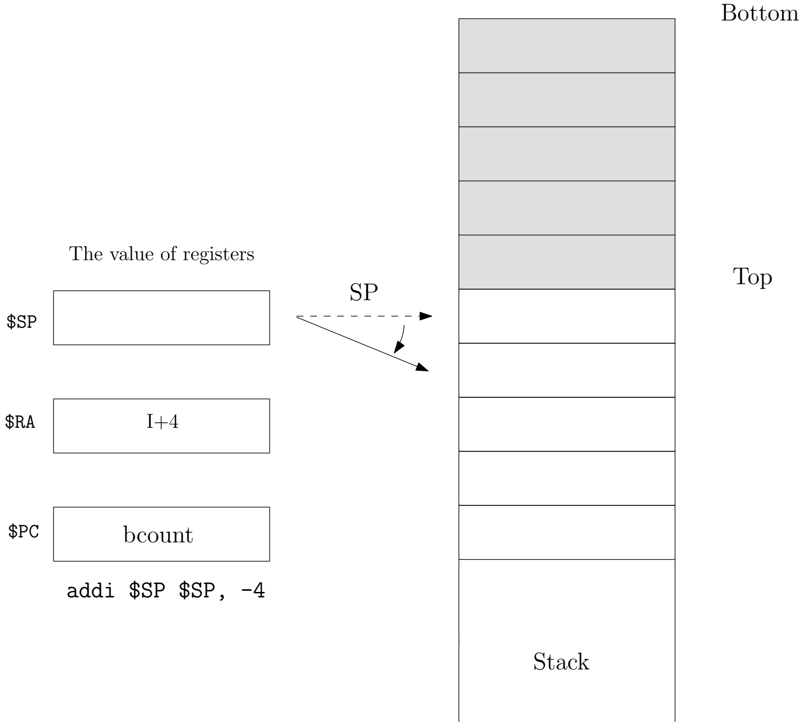
```
          ...

          lw   $RA, 4($SP)      # pop return address off stack
          addi $SP, $SP, 4      # shrink stack
K:        jr $RA                # return to main program


bfind:    ...
          ...
L:        jr $RA                # return to bcount
```

The following figures give an overview of how registers $PC, $RA and $SP, as well as the stack get adjusted during the execution of the previous program. It all starts with $PC having the value I. Pay especially attention to how the stack is used to preserve the content of $RA for when bcount returns to the main program.

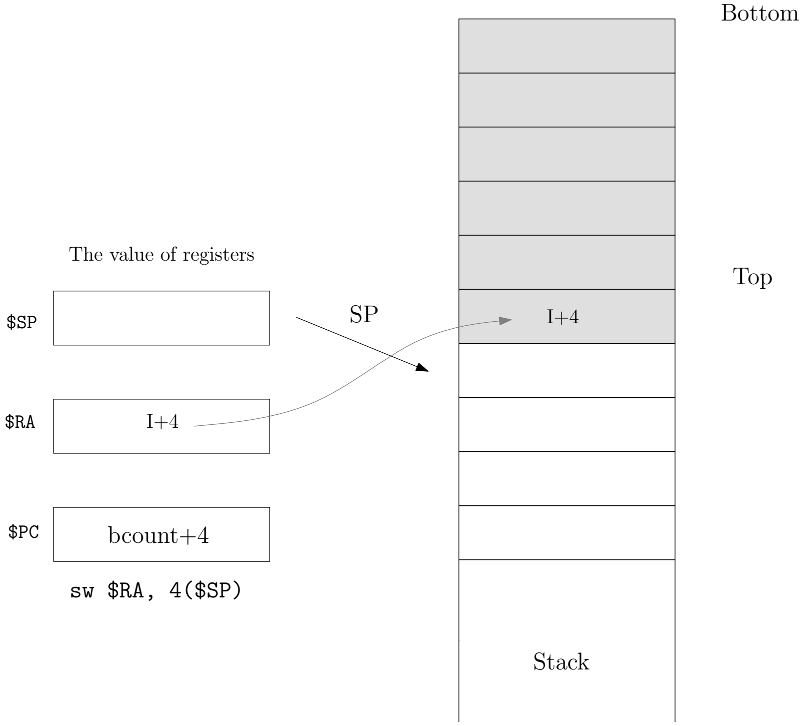
1) Executing instruction at I



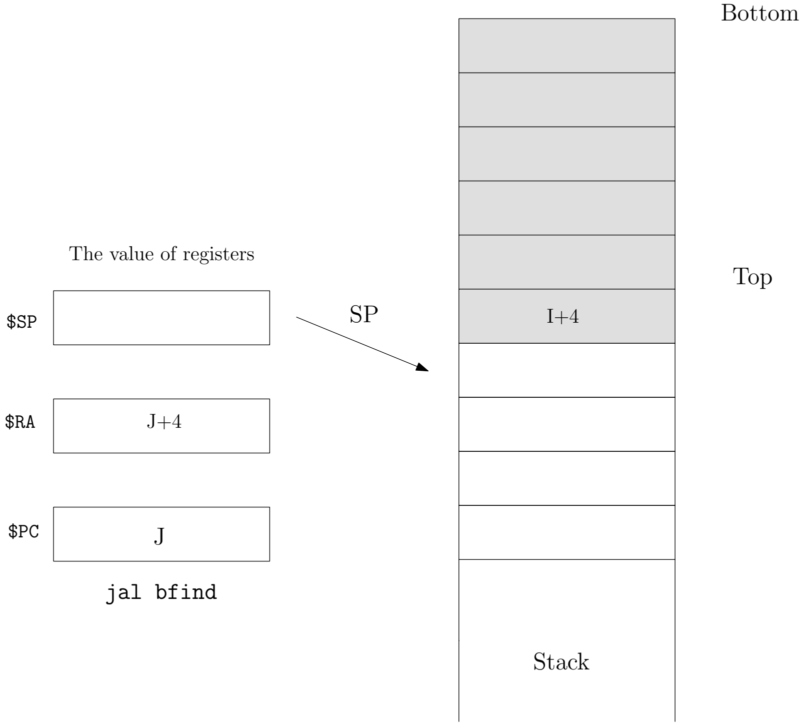
2) Executing instruction at `bcount`



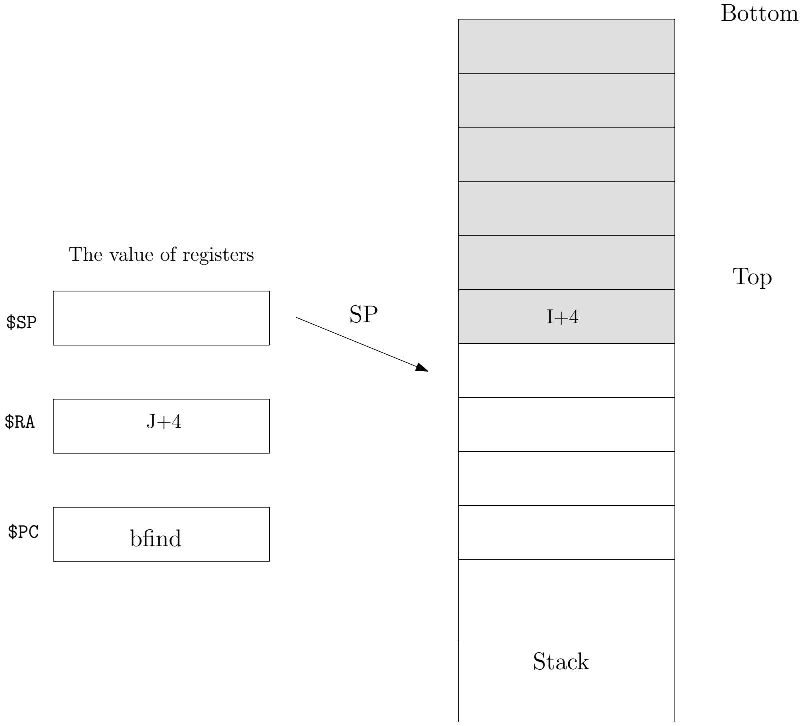
3) Executing instruction at `bcount+4`



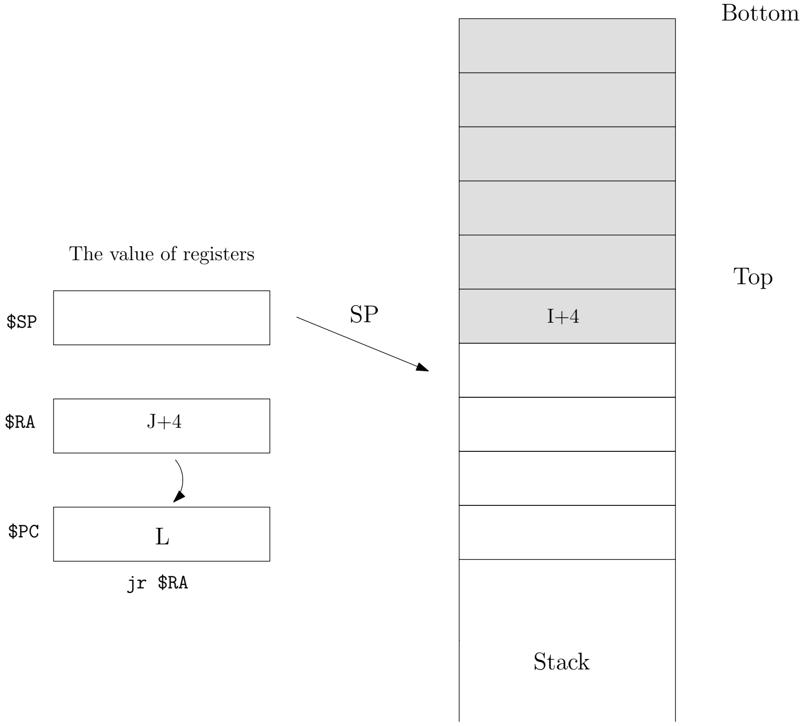
4) Executing instruction at `J`



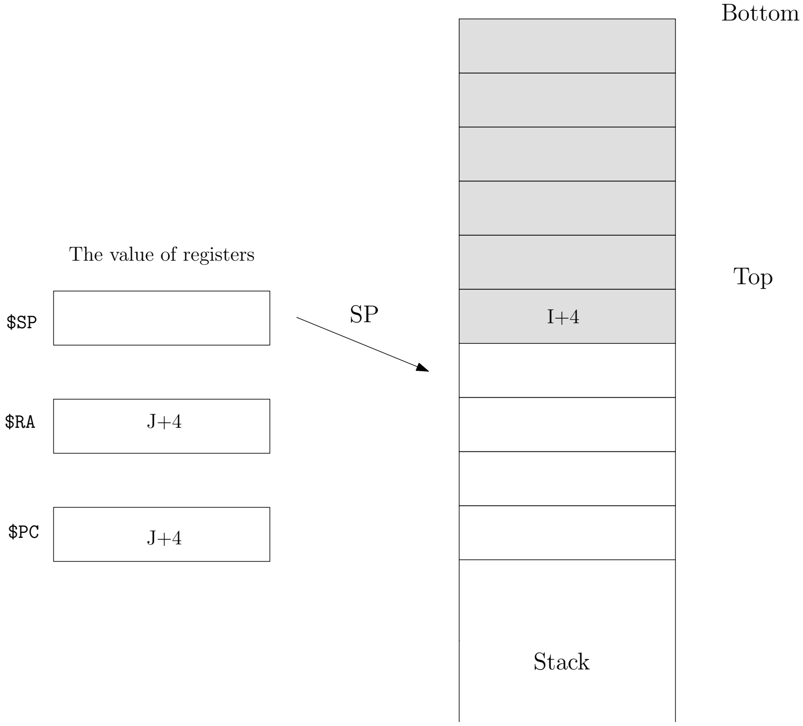
5) Executing instruction at `bfind`



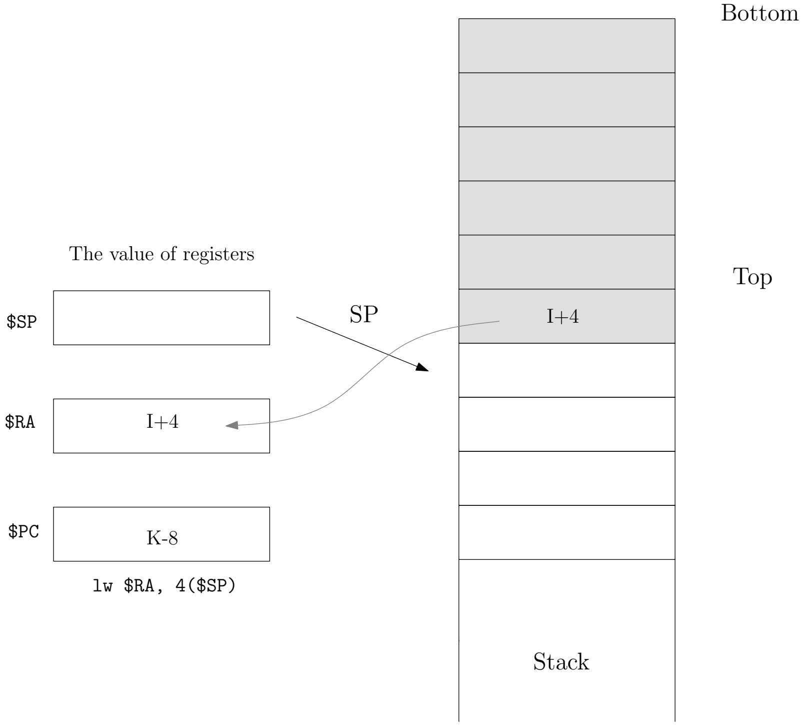
6) Executing instruction at `L`



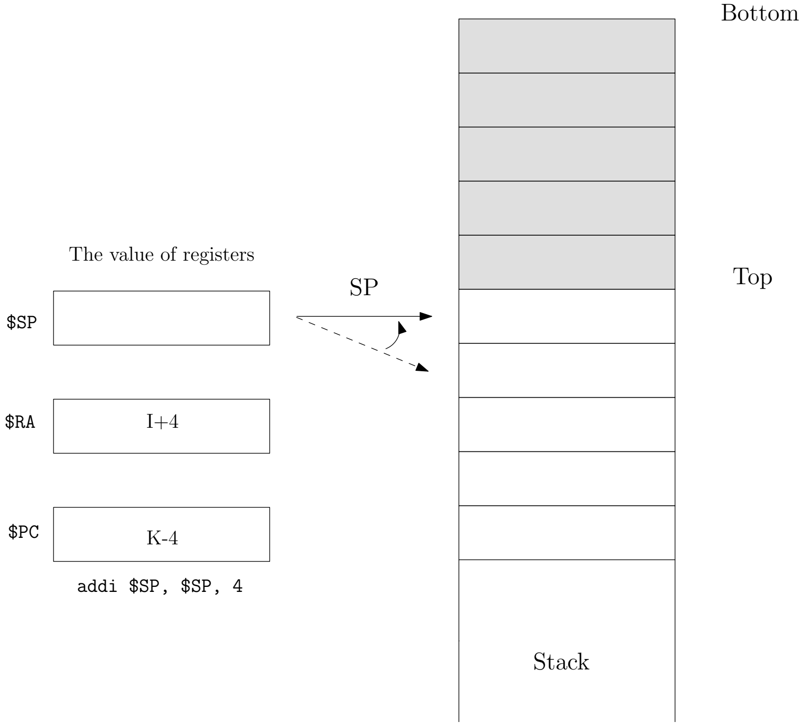
7) Executing instruction at `J+4`



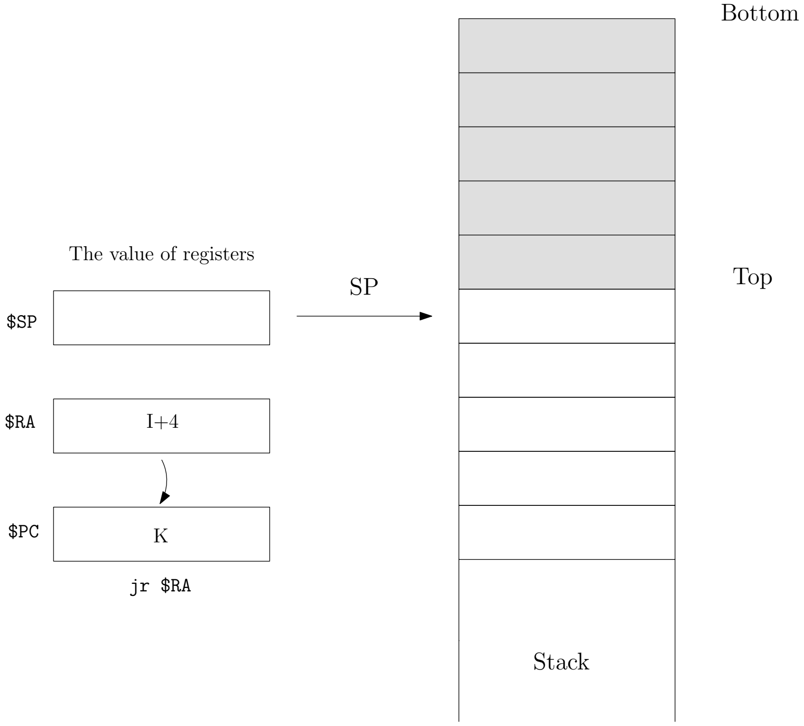
8) Executing instruction at `K-8`



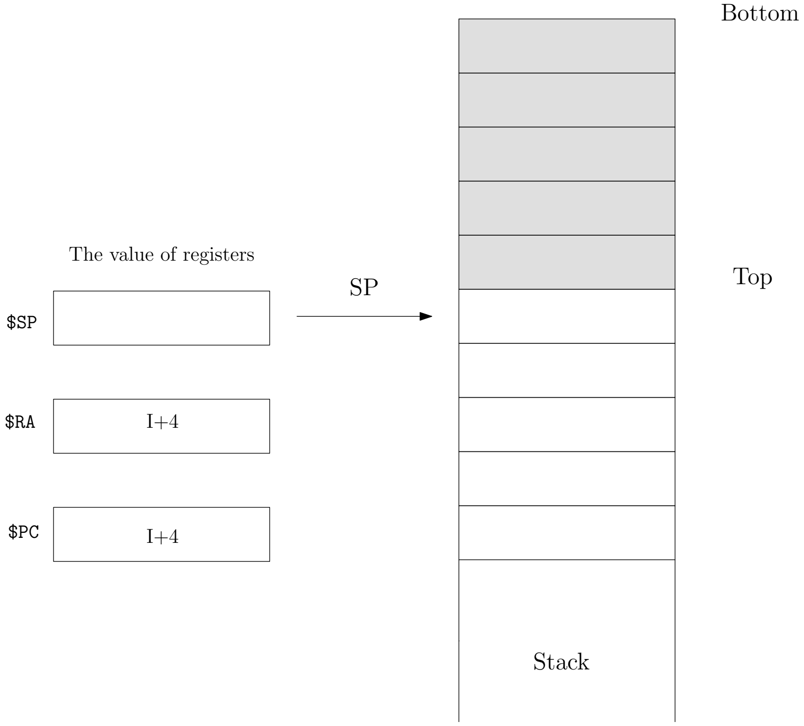
9) Executing instruction at `K-4`



10) Executing instruction at `K`



11) Executing instruction at `I+4`



2.4.4 Formalizing working with procedures more rigorously

So far, we have introduced a series of concepts to support working with procedures in MIPS R2000. Next, we will formalize more rigorously how to work with procedures in MIPS R2000.

Memory organization in MIPS

Below, we give an outline of the conventions that define what part of the memory is used for what purpose. MIPS divides memory into different segments.

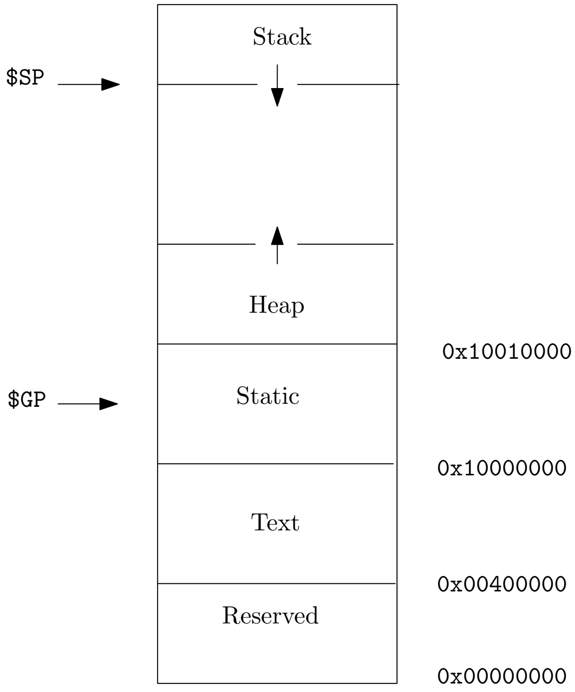


Figure 2.2: Memory in MIPS R2000.

1. **Stack**

   The stack starts at the highest available memory address and grows down from there. The stack contains *automatic* variables, which are local to procedures and discarded upon return from procedures.

2. **Reserved segment**

   The reserved segment is all the way at the bottom of the memory, from address 0x00000000 to 0x00400000. This segment is reserved for the OS (operating system).

3. **Text segment**

   The text segment starts at 0x00400001 and continues until 0x10000000. This segment is used to store MIPS machine code, i.e., the program.

4. **Static data segment**

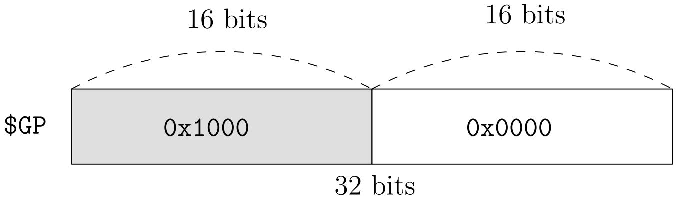
The static data segment spans the memory addresses from `0x10000001` until `0x10010000`. It stores static data, which consists of constants and static or global variables (which exist globally, across any procedure entry or exit, and have a lifetime that is the entire program execution), of fixed size (i.e., size that is known a priori). The special register `$GP` (**global pointer**) is pointing to the middle of the static data segment, at address `0x10008000`. That way, data in the static data segment can be accessed using relative addressing, i.e., by using positive or negative 16-bit offsets from `$GP`. Indeed, to load a value from the static data segment into register `$R`, we can use

```
lw $R, offset($GP)
```

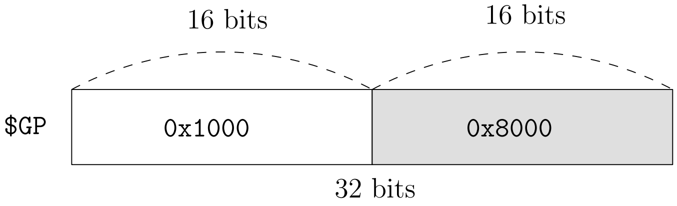
where `offset` is encoded by 16 bits. Thus, `offset` can have a value from -2^{15} to $2^{15}-1$, which exactly covers the entire static data segment, from `0x10000001` until `0x10010000`.

To store the 32-bit word `0x10008000` into the register `$GP`, we cannot use a single MIPS R2000 instruction, since instructions can only be 32 bits long. Therefore, the instruction `lui` (**load upper immediate**) is used. `lui` allows to set the upper 16 bits of a register to a certain value, while setting the lower 16 bits of the register to 0. A subsequent instruction can then specify the lower 16 bits of the register. For example, to store `0x10008000` into `$GP`:

`lui $GP, 0x1000` ⇒



`ori $GP, $GP, 0x8000` ⇒



Note that `ori` (**or immediate**) is used to specify the lower 16 bits of the register. The instruction `ori` will pad the 16-bit immediate value with 16 zero bits on the left, to obtain a 32-bit word before performing a bitwise `or`.

5. **Heap: dynamic data segment**

   The dynamic data segment or heap starts right above the static data segment. The heap contains global variables which grow and/or shrink in size during the program's execution, such as linked lists in C language.

   During the program's execution, the stack and heap can grow towards each other. The programmer needs to make sure they do not grow "into one another". When the distance between the top of the stack and the top of the heap is small, at all times during a program's execution, memory is used maximally efficient.

Precisely defined caller and callee tasks

We now rigorously define which are the organizational tasks that need to be performed when a procedure is called, to ensure no data is lost after the procedure call. As depicted below, the **caller** is the procedure (possibly the main program, which, in itself, can be considered as a procedure) that calls another procedure, while the **callee** is the procedure that is being called, to execute a specific task. As discussed before, when procedures are nested, argument registers, the return address register and other registers might get overwritten. In this section, we rigorously define the organizational tasks that caller and callee are responsible for, at a procedure call, to avoid such errors and data loss. We will see how the stack plays an important role, to save and restore register values in an organized manner.

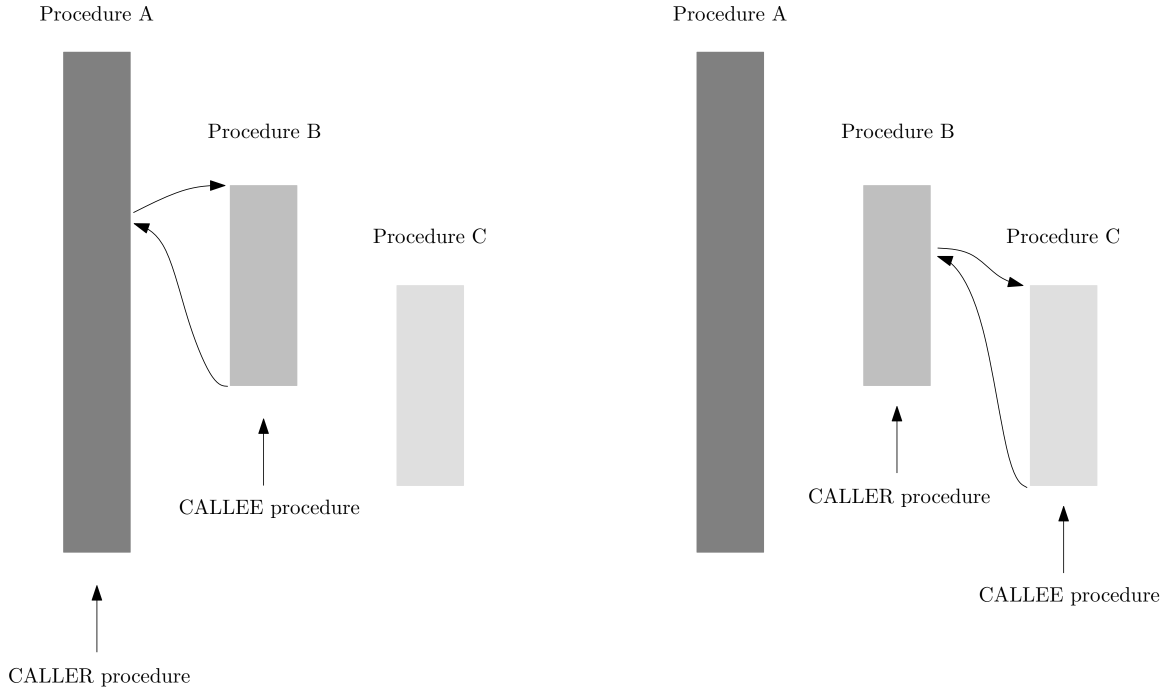


Figure 2.3: Defining the caller and callee.

A) Caller's organizational tasks, before calling the callee

1. Push the values of any or all of `$a0, ..., $a3` on the stack, if the caller still needs the corresponding register's value after the callee returns. Please note that some of the registers `$a0, ..., $a3` might currently hold the arguments which, at some point in the past, were passed to the *caller* procedure, when the caller itself was called by a higher-level procedure. Those registers' values might get overwritten by the caller (when calling the callee) or the callee (in case the callee would call other, nested procedures). See Figure 2.4.

2. Pass arguments from the caller to the callee using the argument registers `$a0, ..., $a3`. If there are more than 4 arguments, push the extra arguments on the stack. See Figure 2.5.

3. Push **temporary registers** `$t0, ..., $t9` on the stack. The temporary registers are 10 out of the 32 CPU registers which can be used by the program, to store any value. They are called "temporary" because the caller cannot rely on their values being preserved when the callee returns. Indeed, if the caller wants to preserve the values of any or all of `$t0, ..., $t9`, it has to push the corresponding values on the stack, before calling the callee, and pop them off the stack and restore the registers' values when the callee returns. So, it is the caller's responsibility to save any or all of the temporary registers at a procedure call, if they need to be preserved. See Figure 2.4.

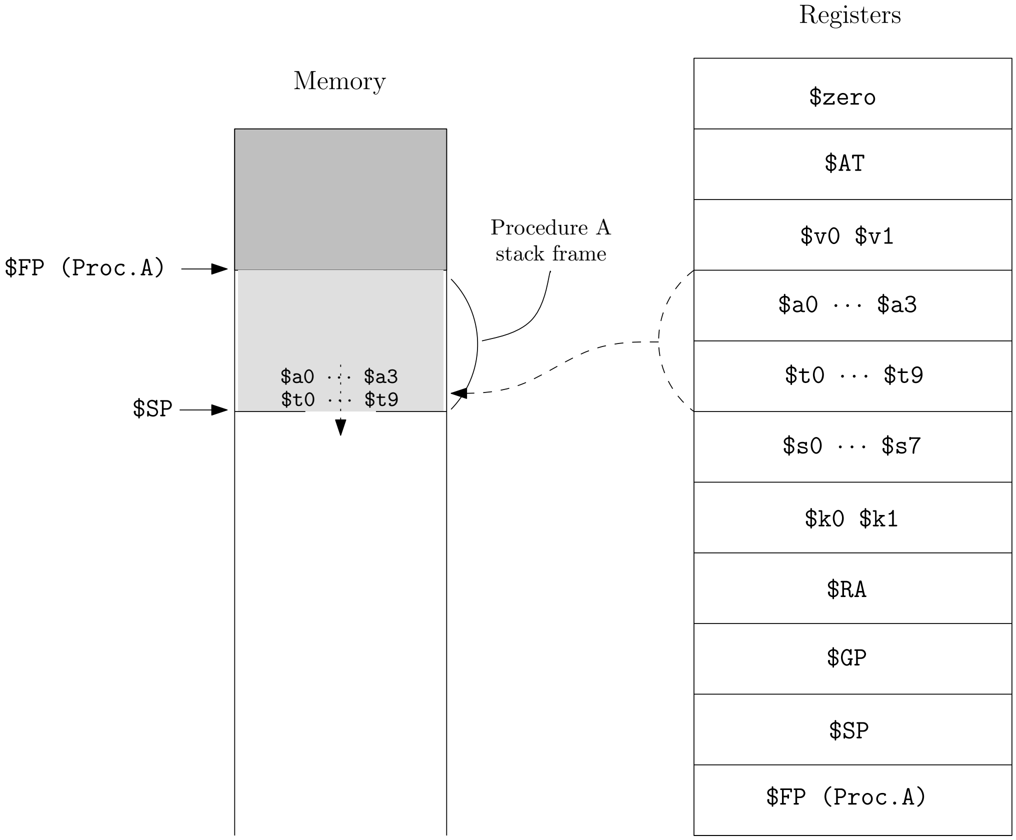


Figure 2.4: Caller A should store any or all of `$a0, ..., $a3` or `$t0, ..., $t9`, if needed.

4. Finally, execute a `jal` instruction, to jump to the callee (i.e., load the address of the callee's first instruction into `$PC`), after linking (i.e., storing the return address in `$RA`).

Stack frame. The segment of the stack that contains all of a procedure's "data" (registers that it decides to save on the stack, variables local to the procedure, etc.) is called the **stack frame**. We know that `$SP` points at the first available memory location, for the stack, so it points to the end of the current procedure's stack frame. The dedicated register `$FP` is a register that points to the beginning of the current procedure's stack frame, as depicted in Figure 2.4. This offers a static, i.e., stable, base address within a procedure, to point to a procedure's local variables, saved on the stack (allowing to refer to each such variable using a fixed offset, from `$FP`). Using `$SP` for this purpose would require dynamically adjusting the offset throughout the procedure, since `$SP` changes dynamically when the stack frame of a procedure grows or shrinks (for example, by pushing on or popping off data). That would make it harder to debug the procedure.

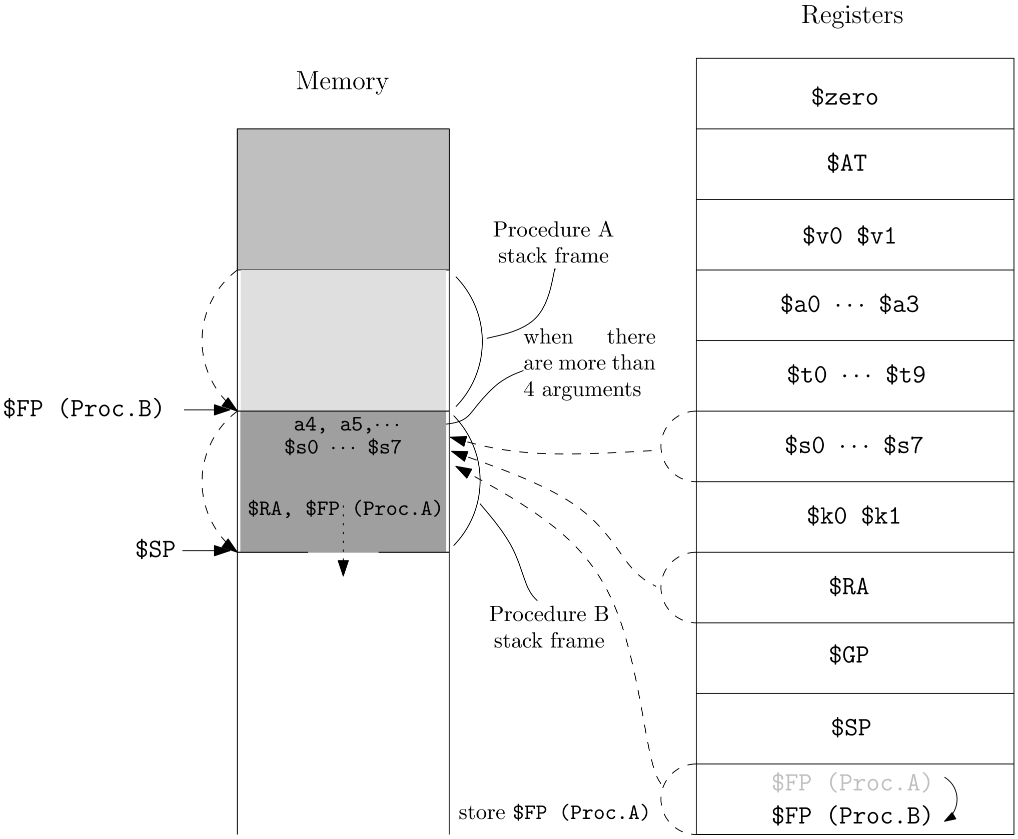


Figure 2.5: Callee B should store any or all of `$s0, ..., $s7`, `$RA` and `$FP`, if needed.

B) Callee's organizational tasks, before executing its principal task

1. Allocate space on the stack to create its initial stack frame. This is done by incrementing `$SP` and `$FP`. To create an initial stack frame of 8 words (32 bytes): `SP = SP - 32`. Adjusting `$FP` is done in Step 5, after the current value of `$FP` has been saved on the stack (in Step 4). See Figure 2.5.

2. Push any or all of the **saved registers** `$s0, ..., $s7` on the stack. The saved registers are 8 out of the 32 CPU registers which can be used by the program, to store any value. They are called "saved" because it is the callee's responsibility to save those registers, if and before using and altering them, so the caller can rely on their values being preserved when the callee returns. Thus, if the callee wants to use and alter any or all of `$s0, ..., $s7`, it has to push the corresponding values on the stack, when it gets called, and pop them off the stack and restore the registers' values before it returns to the caller. The temporary registers `$t0, ..., $t9`, on the other hand, can be altered by the callee without saving them: indeed, it's caller's responsibility to store them, if their values need to be preserved. See Figure 2.5.

3. Push `$RA` on the stack (this can be omitted if the callee is a leaf procedure). See Figure 2.5.

4. Push `$FP` on the stack (before altering it in Step 5). See Figure 2.5.

5. Make `$FP` point to the beginning of the callee's stack frame: `FP = SP + 32`. See Figure 2.5.

While procedure `B` executes its principal task, it may store some local variables onto its (dynamically growing and shrinking) stack frame, as depicted in Figure 2.6.

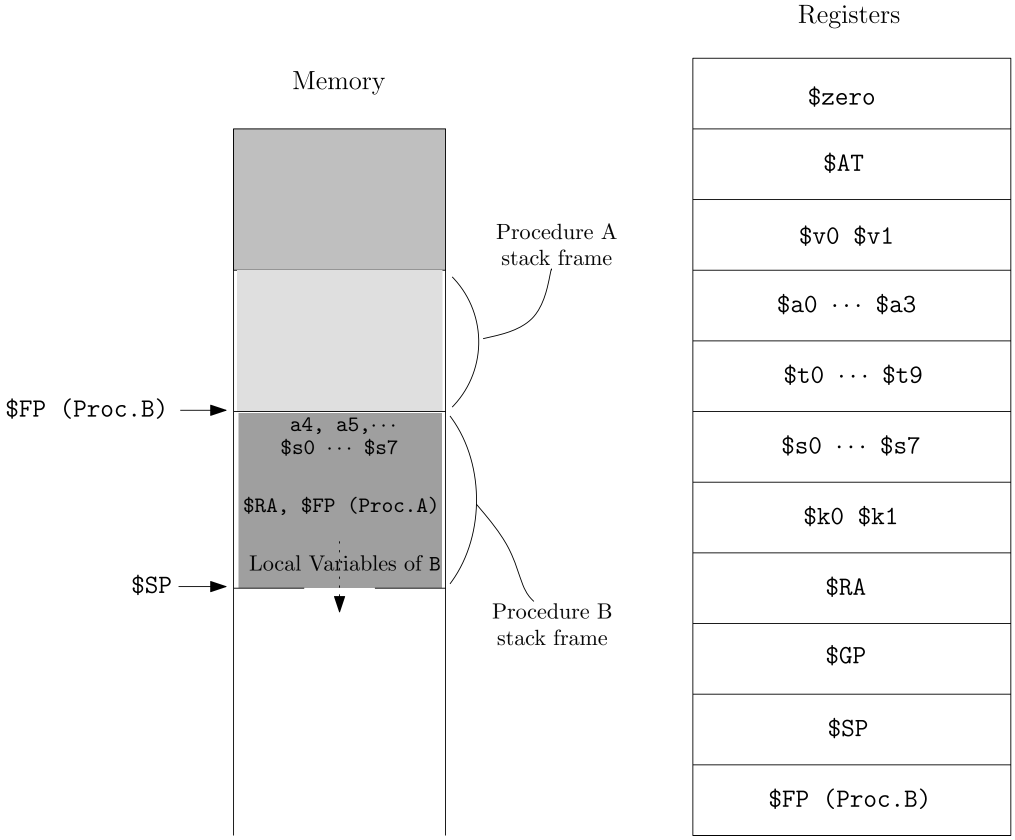


Figure 2.6: Stack frame of `B` while executing its principal task.

C) Callee's organizational tasks, before returning to the caller

1. Place the results that need to be returned to the caller, if any, in the value registers `$v0, $v1`. See Figure 2.7.
2. Restore all of `$s0, ..., $s7, $RA, $FP` that were stored on the stack upon procedure entry. See Figure 2.7.
3. Pop the stack frame by restoring the stack pointer to its value at procedure entry (assuming all local variables have already been popped off): `SP = SP + 32`. This is exactly what the caller expects to see, where nothing on its own stack frame, between the restored `$FP` and `$SP` has been affected. See Figure 2.7.
4. Return control to the caller: `jr $RA`.

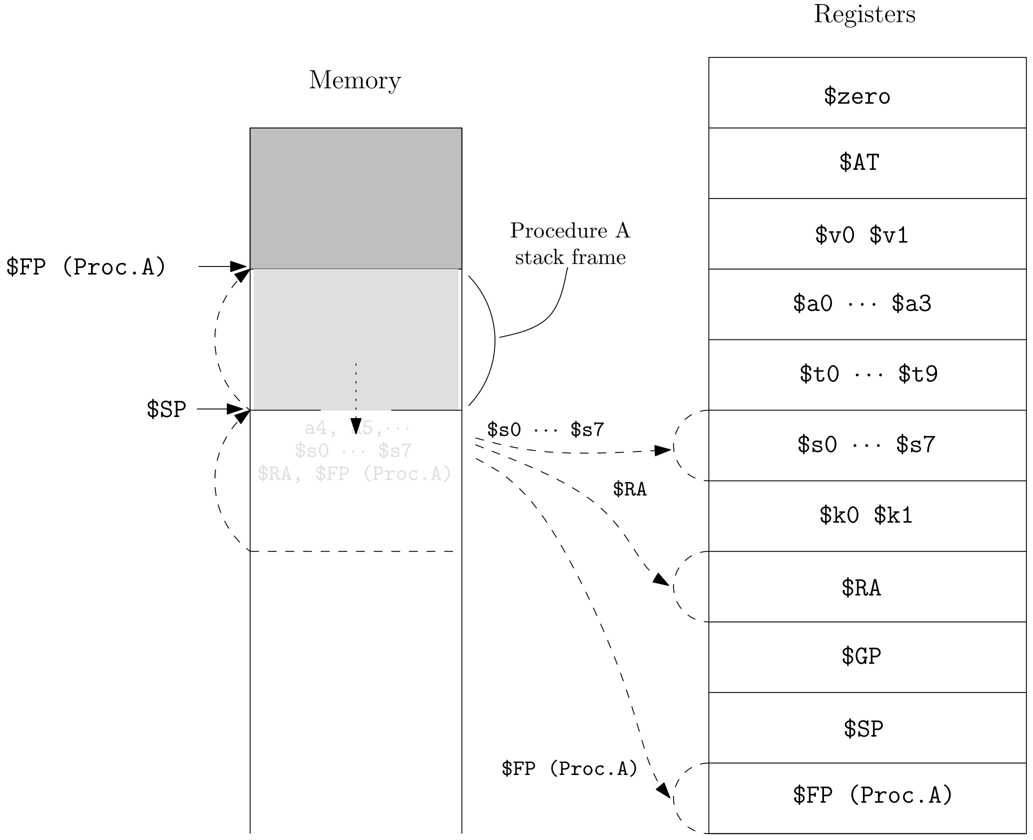


Figure 2.7: Restore all of `$s0, ..., $s7, $RA, $FP` and `$SP` that were altered by the callee.

.

Registers `$zero, $AT, $k0` and `$k1`. By now, we have specified the purpose of 28 out of the 32 CPU registers. `$zero` is a register that is hard-wired to 0, i.e., it contains the value 0 at any time. The three remaining registers, `$AT, $k0` and `$k1` have a designated purpose and will be described later in the text.

Here is a summary of the register usage we have covered so far.

Name	Number	Usage
`$zero`	0	Constant 0
`$AT`	1	Reserved for assembler
`$v0-1`	2-3	Return values from procedures
`$a0-3`	4-7	Arguments 1-4 for procedures
`$t0-7`	8-15	Temporary registers
`$s0-7`	16-23	Saved registers
`$t8-9`	24-25	Temporary registers
`$k0-1`	26-27	Reserved for OS kernel
`$GP`	28	Pointer to static data segment
`$SP`	29	Stack Pointer
`$FP`	30	Frame Pointer
`$RA`	31	Return Address

2.5 From C program to machine code

To execute a C program, it is first translated into low-level assembly language by a **compiler**. Alternatively, a programmer could also directly write (often even more efficient) code in assembly language. Next, the low-level assembly code, i.e., a human readable form of what the machine really understands (bits and bytes), needs to be translated into machine language, i.e., a sequence of 32-bit machine instructions. This is done by the **assembler**. Finally, the executable, machine language program (stored as a file on the hard drive) is loaded into memory by the **loader**, and execution started.

To help the programmer to write assembly programs more easily and transparently, as well as the assembler, to translate assembly into appropriate machine code, the following concepts are supported by the assembler, although they are not actual assembly instructions: **pseudo instructions**, **labels** and **directives**.

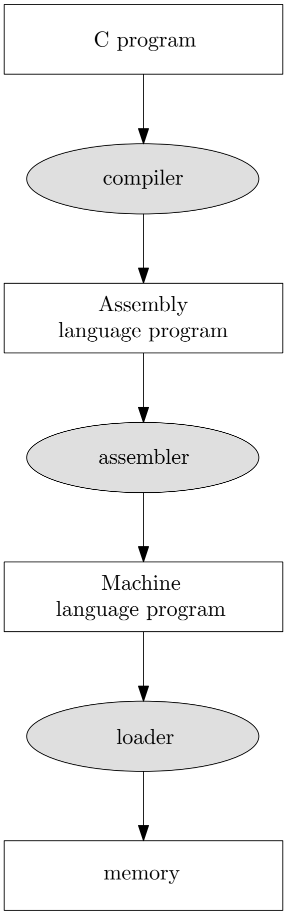


Figure 2.8: Translating and loading programs.

2.5.1 Pseudo instructions

Pseudo instructions are defined in assembly language, to give the programmer a richer set of instructions to work with, however, they do not correspond to an actual machine language instruction that the hardware (CPU) can implement directly. Therefore, the assembler first translates them into a sequence of one or more assembly instructions that each correspond to a real machine instruction (that the CPU can execute). Next, that (sequence of) assembly instruction(s) is translated into a sequence of 32-bit instructions in machine language, for the CPU to execute.

Example 2.5.1

The pseudo instruction `move` (to copy the content of one register into another) can be translated into machine language after first translating it into an equivalent assembly instruction:

$$\texttt{move \$8, \$18} \iff \texttt{add \$8, \$zero, \$18}$$

Note that `move` only has two argument registers. Pseudo instructions can violate the standard MIPS format.

Example 2.5.2

The pseudo instruction `blt` (branch if less than) is first translated, by the assembler, into:

```
blt $19, $20, L1  ⟺  slt $1, $19, $20
                     bne $1, $zero, L1
```

Next, the assembler translates the `slt` and `bne` instructions into two 32-bit machine language instructions.

Example 2.5.3

The pseudo instruction `ble` (branch if less than or equal to) can be translated into machine language as

```
ble $19, $20, L1  ⟺  slt $1, $20, $19
                     beq $1, $zero, L1
```

In the previous two examples, the assembler translated one pseudo instruction into a sequence of two machine language instructions, using an extra register `$1`, to save a temporary value. If `$1` were already used by the program, for other purposes, this translation could cause data loss. Therefore, the assembler is required to use the designated register `$AT` for such purposes: a temporary register reserved for use by the assembler only, e.g., to translate pseudo instructions. The `blt` example will thus be translated as

```
blt $19, $20, L1  ⟺  slt $AT, $19, $20
                     bne $AT, $zero, L1
```

2.5.2 Labels

Labels, defined before as a “name” followed by “:”, are used to specify a memory location in assembly language, without the need to know the exact memory address. The assembler will make sure that every label is translated into the right memory reference (a real, absolute or relative memory address).

2.5.3 Assembler directives

Assembler directives are defined as a “.” followed by a “name” and are used to direct the assembler on how to exactly translate a program. A directive is not translated into a specific machine language instruction. Here are some directives:

- `.text`: tells the assembler that what follows is a code segment that contains the machine language source code for one or more procedures that execute specific tasks, which needs to be stored in the text segment in memory.

- `.data`: tells the assembler that what follows is a code segment that contains the binary representation of some data used by the program, to be stored in the data segment in memory.

- `.globl L`: tells the assembler that the label `L` is a "global" label, i.e., that it can be referenced from files other than the one in which it is defined (as opposed to "local" labels, the default, which can be used only within the file in which they're defined). So, the assembler is directed to make `L` visible to other files.

- `.extern L`: `L` is declared as a global label in another file

- Data size directives:

  - `.byte b1, ..., bn`: store the n values `b1, ..., bn` in successive bytes of memory
  - `.word w1, ..., wn`: store the n values `w1, ..., wn` in successive words of memory
  - `.float f1, ..., fn`: store the n values `f1, ..., fn` as single precision floating point numbers, in successive memory locations
  - `.double d1, ..., dn`: store the n values `d1, ..., dn` as double precision floating point numbers, in successive memory locations

- Data alignment directive: `.align n` aligns the next data on 2^n byte boundary. For example

  - `.align 2`: aligns the next value on the next word boundary
  - `.align 0`: no alignment (aligns the next data on the next byte boundary)

- String directives

  - `.ascii`: ASCII string
  - `.asciiz`: null terminated ASCII string
  - `.space` n: allocate n bytes of space in the data segment

In assembly language, decimal numbers are represented in decimal notation `256, 1098`. Hexadecimal numbers are represented using `0x` followed by the actual hexadecimal representation, e.g., `0x10C`. To represent strings, the C language convention is followed: strings are enclosed in double quotes and one can use special characters like `\n`, to represent a newline, `\t` for a tab and `\"` for a quote.

2.6 SPIM

SPIM (from MIPS, spelled backwards) is a self-contained software simulator that reads and executes MIPS R2000 assembly language programs. SPIM also provides a simple debugger and a minimal set of operating system services. The following are some essential features:

- `clear`: clear registers and memory
- `load`: load program
- `run`: run program
- `step`: single-step program (used to debug)
- `breakpoint`: set/delete breakpoints (used to debug)

SPIM also provides system services, through the `syscall` instruction. To request a service, one loads the system call code into `$v0`, the arguments into `$a0, ..., $a3`, call `syscall` and, if any, retrieve the return values from `$v0, $v1`.

Service	System call code	Arguments
Print int	1	`$a0` = integer
Print string	4	`$a0` = pointer to null-terminated string
Exit	10	

For example, to print the integer 5:

```
li  $a0,  5      # load integer 5 into $a0
li  $v0,  1      # system call code to print integer
syscall          # system call
```

2.7 Example

In this section, we will write and analyze a program to compute $n!$. Looking at the computation of 10!, as an example, we note that

$$\begin{aligned} 10! &= 10 \cdot 9 \cdot ... \cdot 1 \\ &= 10 \cdot 9! \end{aligned}$$

In general, $n! = n \cdot (n-1)!$. Assume that `fact(n)` is a procedure that returns $n!$, for any positive integer n. To implement the procedure `fact()`, we could use the previous property. Assuming `fact()` gets called with an argument `n`, we could do the following: call `fact()` with argument `(n-1)` and multiply the result with `n`, to obtain the correct value for `fact(n)` by returning `n · fact(n-1)`. So, the procedure `fact()` calls itself recursively. This will eventually lead to a solution since the recursive calls of `fact()` involve a smaller argument, which simplifies the problem at every step. Eventually, `fact()` will be called with argument 0. At that point, it returns the value 1, the recursive calls stop and the nested products are computed to return the result.

This is illustrated with the computation of `fact(10)`, below, using this approach: after calling `fact()` with argument 10, the procedure `fact()` calls itself 10 more times, recursively;

eventually, 1 is returned as the result for fact(0) and the computation of the recursive products is wrapped up.

$$
\begin{aligned}
\texttt{fact(10)} &= 10 \cdot \texttt{fact(9)} \\
&= 10 \cdot 9 \cdot \texttt{fact(8)} \\
&= 10 \cdot 9 \cdot 8 \cdot \texttt{fact(7)} \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot \texttt{fact(6)} \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot \texttt{fact(5)} \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot \texttt{fact(4)} \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot \texttt{fact(3)} \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot \texttt{fact(2)} \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot \texttt{fact(1)} \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot \texttt{fact(0)} \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 6 \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 24 \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 120 \\
&= 10 \cdot 9 \cdot 8 \cdot 7 \cdot 720 \\
&= 10 \cdot 9 \cdot 8 \cdot 5040 \\
&= 10 \cdot 9 \cdot 40320 \\
&= 10 \cdot 362880 \\
&= 3628800
\end{aligned}
$$

According to this principle, the following C code computes 10! and prints the result on the screen:

```
main()
{
  printf("The factorial of 10 is %d\n", fact(10));
}

int fact (int n)
{
  if (n < 1)
    return 1;
  else
    return (n * fact(n-1));
}
```

The corresponding assembly program is given by

```
#Line
 0       .text                     # machine code stored in text segment in memory
 1       .extern printf            # will call printf procedure defined in other file
 2       .globl main               # main procedure can be called from other files
 3 main: addiu $sp, $sp, -32       # begin callee organizational tasks; stack frame
 4       sw    $ra, 20($sp)        # $ra stored on stack frame
 5       sw    $fp, 16($sp)        # $fp stored on stack frame
 6       addiu $fp, $sp, 32        # end callee organizational tasks; setup $fp
 7
 8       li    $a0, 10             # argument for fact procedure call is 10
 9       jal   fact                # call fact
 10
 11      la    $a0, LC             # first argument printf: pointer to format string
 12      move  $a1, $v0            # second argument printf: result of fact
 13      jal   printf              # call procedure printf (external library)
 14
 15      lw    $ra, 20($sp)        # begin callee organizational tasks; restore $ra
 16      lw    $fp, 16($sp)        # restore $fp
 17      addiu $sp, $sp, 32        # end callee organizational tasks; pop stack frame
 18      jr    $ra                 # return to caller
 19
 20      .data                     # data stored in data segment in memory
 21 LC:  .asciiz "The factorial of 10 is %d\n"
 22
 23      .text                     # machine code stored in text segment in memory
 24 fact:addiu $sp, $sp, -32       # begin callee organizational tasks
 25      sw    $ra, 20($sp)
 26      sw    $fp, 16($sp)
 27      addiu $fp, $sp, 32        # end callee organizational tasks
 28
 29      sw    $a0, 0($fp)         # preserve $a0 on stack before recursive call fact
 30
 31      bgtz  $a0, L2             # if n > 0 compute fact(n) as n * fact(n-1) at L2
 32      li    $v0, 1              # else return 1 (for n = 0)
 33      j     L1                  # go to organizational tasks before returning
 34
 35 L2:  addi  $a0, $a0, -1        # compute n - 1
 36      jal   fact                # call fact with argument n - 1
 37      lw    $a0, 0($fp)         # restore original argument, n, from stack
 38      mul   $v0, $v0, $a0       # fact(n) = fact(n - 1) * n
 39
 40 L1:  lw    $ra, 20($sp)        # begin callee organizational tasks
```

```
41      lw    $fp, 16($sp)
42      addiu $sp, $sp, 32        # end callee organizational tasks
43      jr    $ra                 # return to caller
```

The pseudo instruction `la $a0, LC` (line number 11) loads the real (assembler computed) memory address corresponding to the label LC (NOT the contents of that memory location) into the register `$a0`. Indeed, the formatting string is too long to fit entirely into `$a0`, so we have `$a0` point to the location of its first character, in memory (label LC). Also, note that `addiu` is used for memory address computations. More about this in the next chapter.

2.7.1 Procedure `main`

The directive `.text`, in the first line of the assembly code, makes it clear to the assembler that what follows is a code segment, to be stored in the text segment of memory. The label `main`, which represents the memory address of the first instruction of the procedure, needs to be global, so it is visible to the external, higher-level procedure which actually calls `main`, using a `jal main` instruction (this can be a procedure operated by the OS or another external program). When `main` gets called, the frame pointer (`$FP`) is pointing to the starting point of the stack frame of the previous, external procedure, while the stack pointer (`$SP`) is pointing to the first available memory location for the stack, i.e., right below its stack frame, as depicted in Figure 2.9.

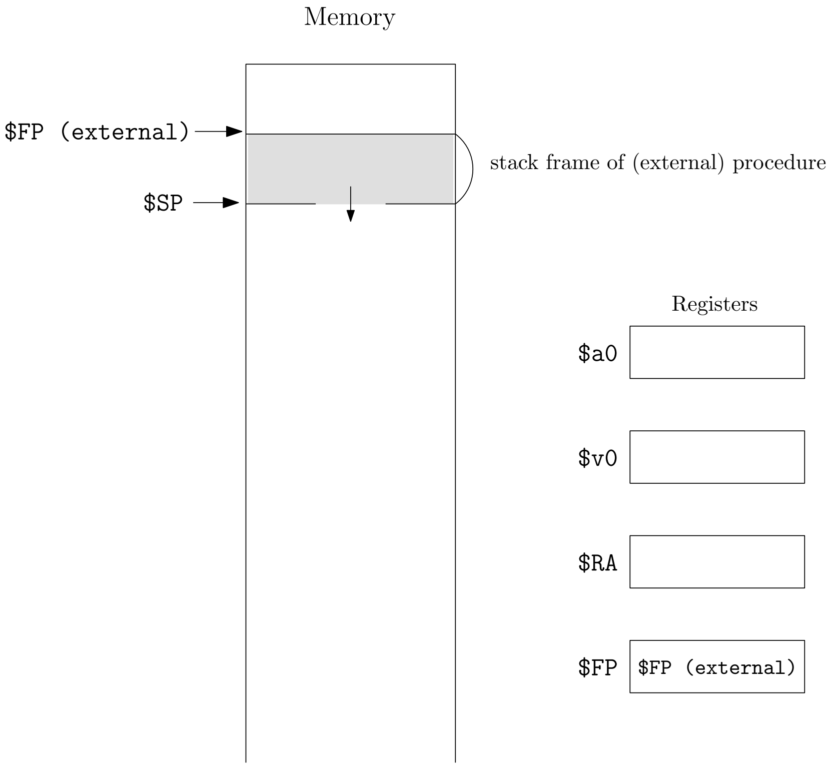


Figure 2.9: Status of the stack when `main` is called.

Once `main` has been called, it first goes through the organizational tasks (lines 3 - 6) that it is responsible for, as *callee*. As discussed in the previous section, it creates its stack frame (`addiu $sp, $sp, -32`) and stores the registers it is responsible to preserve, only `$RA` and `$FP` in this case, on the stack (lines 4 - 5). With the previous value of `$FP` saved on the stack, it can now move `$FP` to the beginning of its own stack frame (line 6), resulting in Figure 2.10.

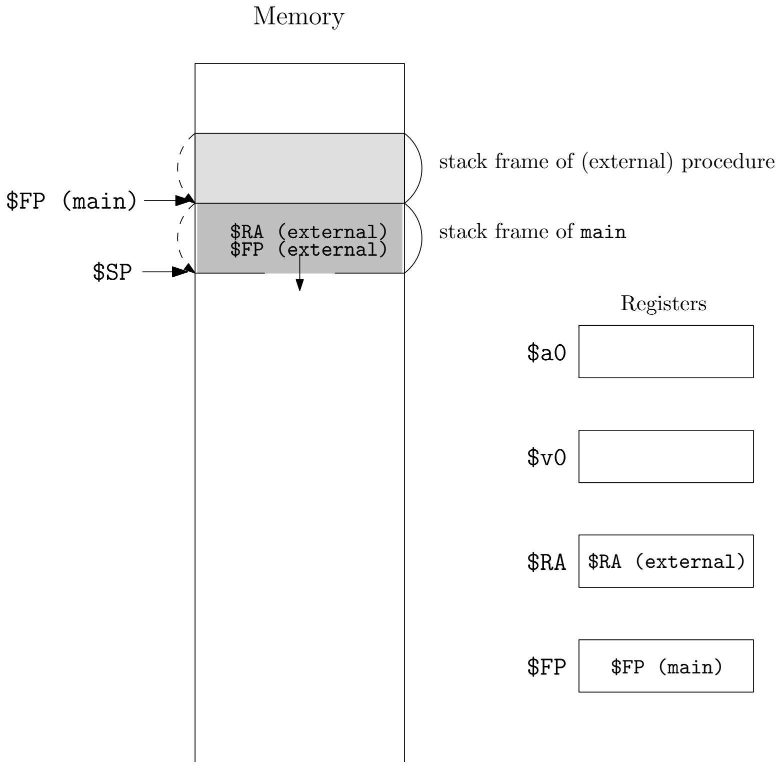


Figure 2.10: Status of the stack, after `main`, as callee, performed its organizational tasks.

Next, code follows to execute the core task, expected from `main`, i.e., compute and print 10!. First, to compute 10!, the procedure `fact` is called (using `jal` in line 9), after storing the appropriate argument (10) in the argument register `$a0` (line 8). For this procedure call, `main` is considered as the *caller* and `fact` as the *callee*. When `fact` returns (line 10), the caller is guaranteed that the saved registers and `$FP` are preserved and that the register `$v0` contains the value `fact` returns, i.e., 10!. During execution, the program counter `$PC` will continue at line 24, after line 9, to execute `fact`. However, when programming, we make abstraction of the actual details of the procedure `fact` and continue directly with the next line of code (line 11), knowing the result returned by `fact` is now in `$v0`.

Second, the value of 10! is printed on the screen by calling the external procedure `printf` (declared external in line 1, using the directive `.extern`) in line 13, after passing on the arguments in lines 11 (the formatting string) and 12 (the value of 10!). In line 11, a pointer to the memory address (label `LC`) where the first character of the formatting string is located, is stored in register `$a0`. On line 21, the actual string is defined and stored in memory, more specifically in the data segment of the memory, as indicated by the `.data` directive in line 20. In line 23, the `.text` directive tells the assembler that what follows is a code segment again and no longer data.

After printing out the result, the core task of `main` has been accomplished and it is supposed to return the control to its caller. Before doing so, it needs to take care of some organizational tasks, as callee (lines 15 - 17): restoring the correct value of the return address, `$RA`, restoring the frame pointer `$FP` to where it was pointing when `main` was called and pop its stack frame. Finally, the stack looks as in Figure 2.11 and, using `jr $ra`, the procedure `main` returns the control to its caller.

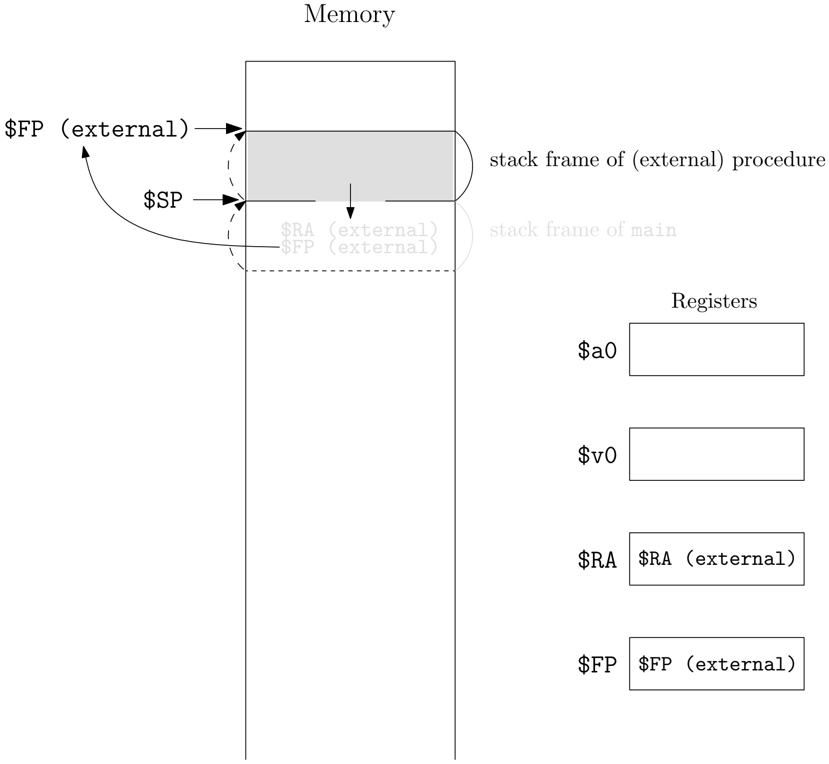


Figure 2.11: Status of the stack right before `main` returns to its caller.

2.7.2 Procedure `fact`

Just like any other procedure, the procedure `fact` (after being called by `main`), first goes through its organization tasks, as callee (lines 24 - 27): creating its stack frame, pushing `$RA` and `$FP` onto the stack and making `$FP` point to the beginning of its stack frame. To distinguish between the different recursive calls of `fact`, we denote a call of the procedure `fact` with argument n as `fact(n)`. If and when `fact` calls itself recursively, the argument register `$a0` will get overwritten with the new argument, i.e., $n - 1$. At that point, it is the responsibility of `fact`, as *caller*, to store the previous value of `$a0`, if it still needs access to that value, later on. Since the latter is indeed the case (see line 38), the value of the argument register `$a0` is pushed onto the stack (line 29), to avoid losing its value. The stack then looks as in Figure 2.12.

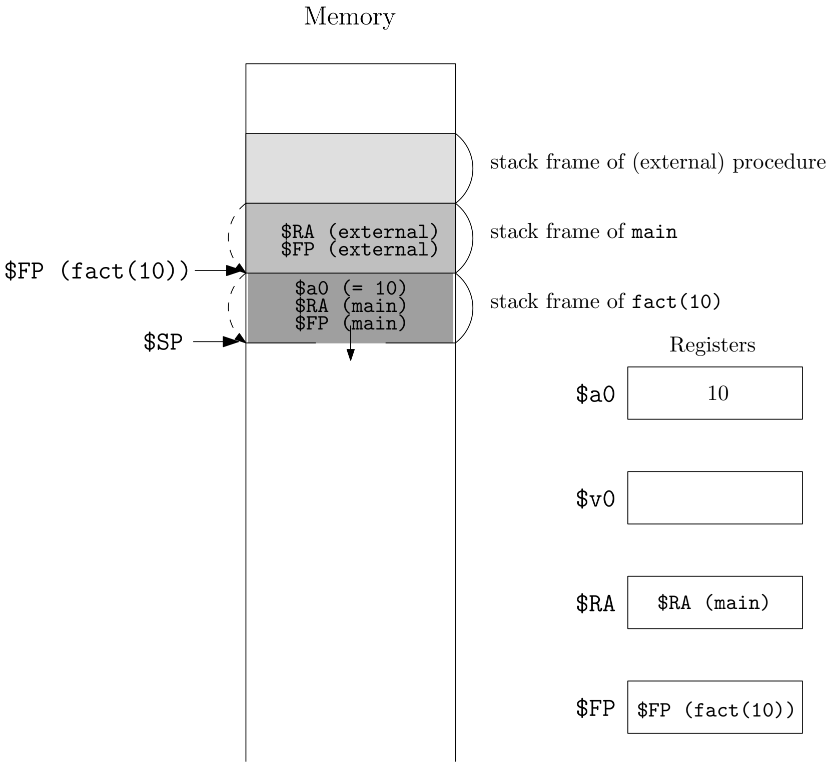


Figure 2.12: Stack after the organizational tasks by `fact(10)`, as callee of `main` and (anticipating to be) caller of `fact(9)`.

The procedure `fact(n)` then proceeds to its core task, for which two cases need to be distinguished: its argument `n` being greater than zero and its argument `n` being smaller than or equal to 0, i.e., equal to 0. In line 31, the argument `n`, stored in `$a0`, is compared to 0 (using the pseudo instruction `bgtz`). If `n` is 0, there is no branching and the next instructions (line 32-33) return the value 1 and then jump to `L1`, for some organizational tasks before returning. This corresponds to `"if(n<1) return 1;"` in the C code.

If `n` is greater than 0, the program branches to `L2` on line 35: after decreasing the value in `$a0` by 1 (line 35), `fact` is called recursively with the decreased argument value in `$a0`, i.e., `n-1`. For this procedure call, `fact(n)` is considered as the *caller* and `fact(n-1)` as the *callee*. When `fact(n-1)` returns (line 37), the caller, `fact(n)`, is guaranteed that the saved registers and `$FP` are preserved and that the register `$v0` contains the value `fact(n-1)` returns, i.e., $(n-1)!$. During execution, the program counter `$PC` will continue at line 24, after line 36, to execute `fact(n-1)`. However, when programming, we make abstraction of the actual details of the procedure call `fact(n-1)` and continue directly with the next line of code (line 37), knowing the result returned by `fact(n-1)` is now stored in `$v0`.

After restoring the original argument of `fact(n)`, by popping it off the stack (line 37), `fact(n)` can now return $n!$ by storing `fact(n-1) * n` in `$v0` (line 38). The procedure then continues at `L1` for some organizational tasks before returning: restoring `$RA` and `$FP` and popping the stack frame. Figures 2.13 to 2.17 illustrate the recursive procedure calls and the building up of stack frames.

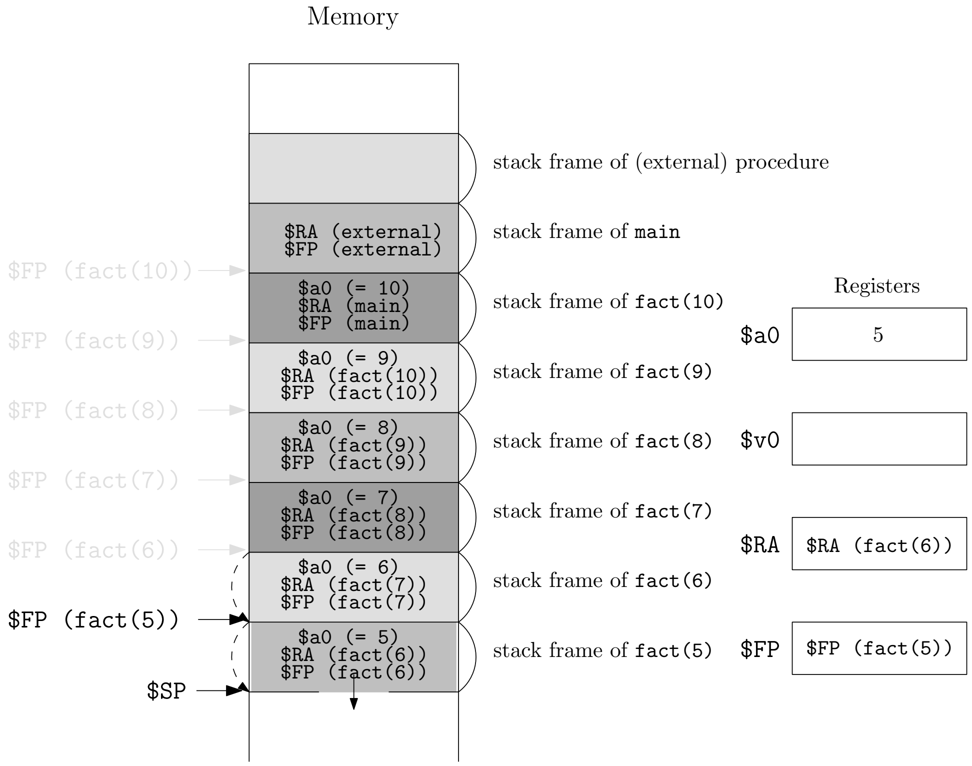


Figure 2.13: Stack and registers before fact(5) starts core task.

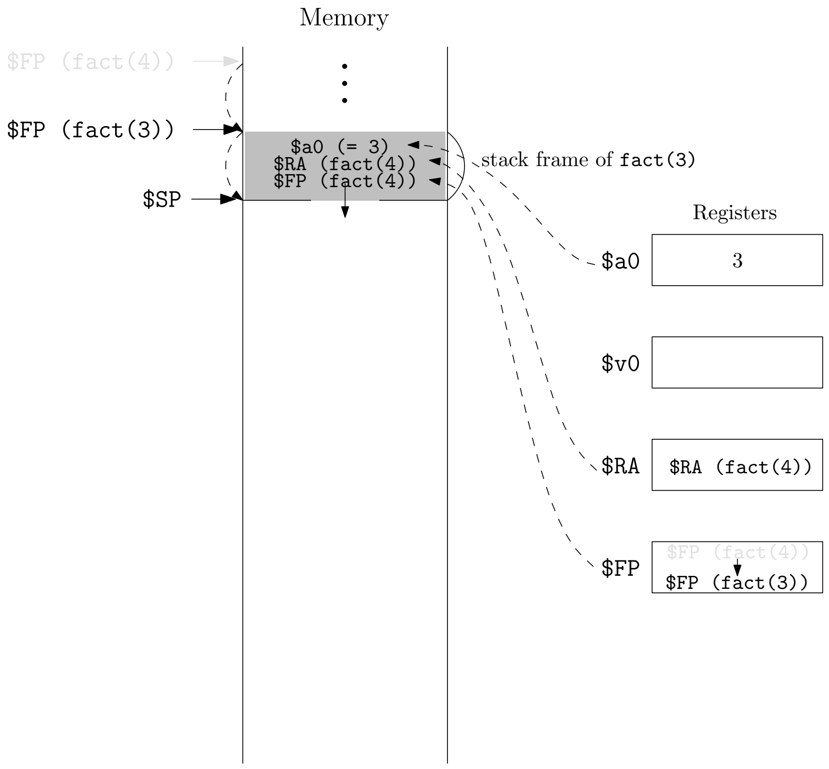


Figure 2.14: Stack frame and registers before fact(3) starts core task.

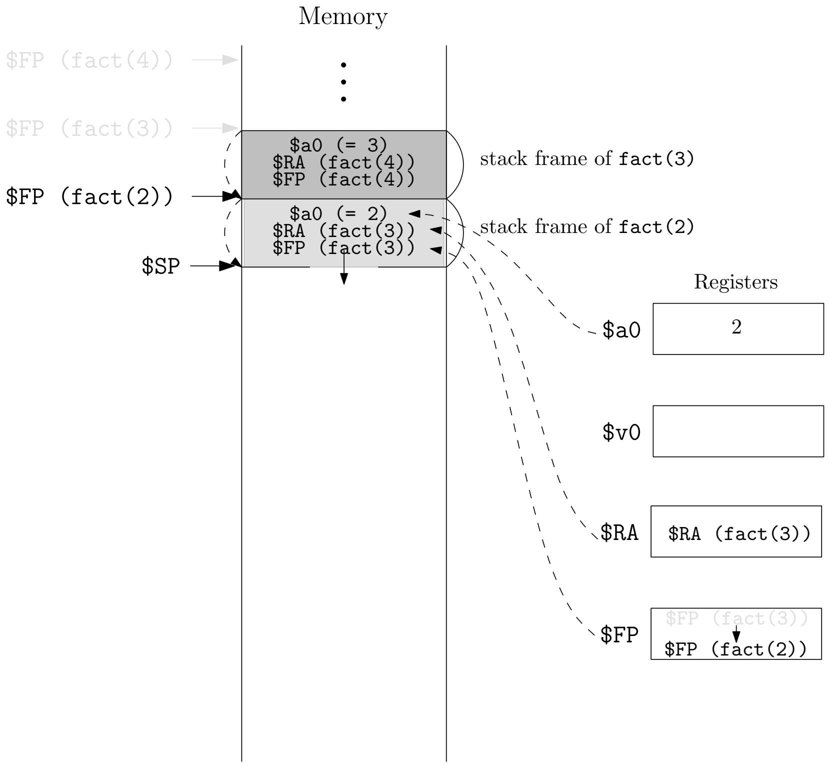


Figure 2.15: Stack frame and registers before `fact(2)` starts core task.

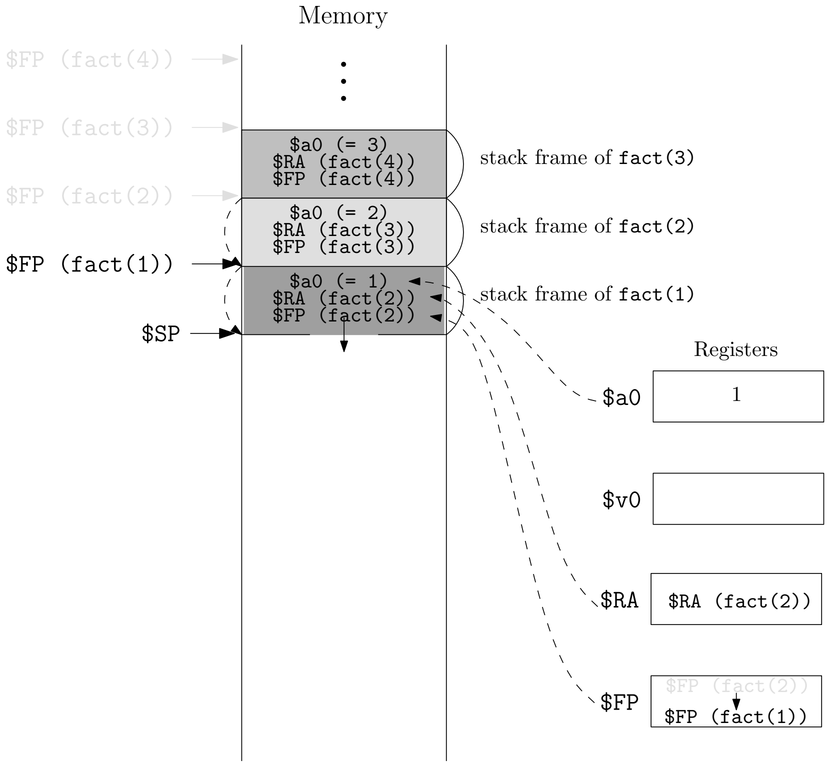


Figure 2.16: Stack frame and registers before `fact(1)` starts core task.

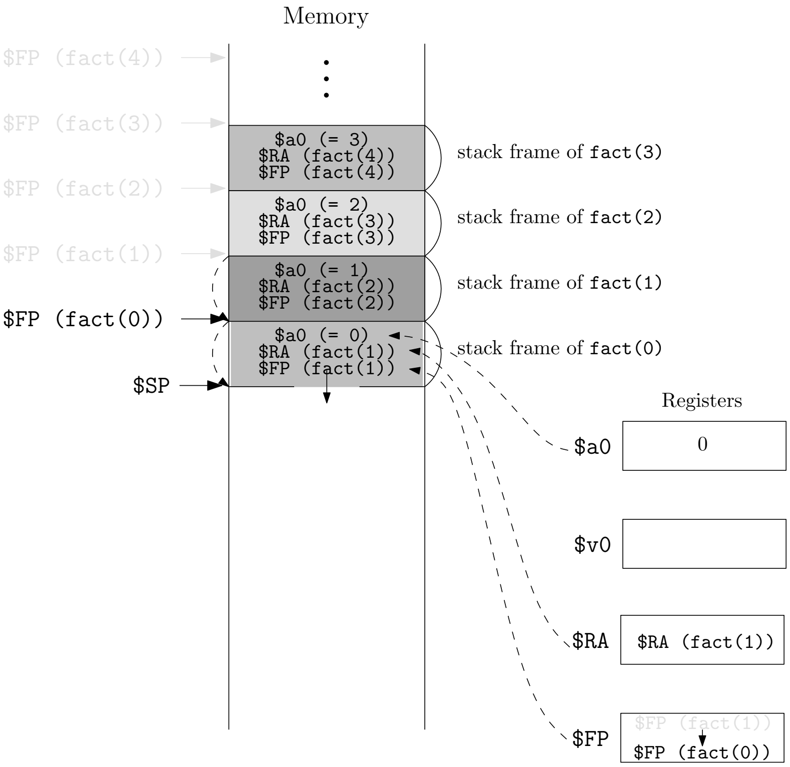


Figure 2.17: Stack frame and registers before `fact(0)` starts core task.

When `fact(0)` is called, the branching instruction in line 31 doesn't branch and the value 1 is returned in the register `$v0`. Jumping to `L1` then executes lines 40-43, which restore `$RA (fact(1))` (not really needed for `fact(0)`) and `$FP (fact(1))`, pop the stack frame and return the control to `fact(1)`. `fact(1)` then continues execution on line 37, returns `1*fact(0)` in `$v0`, and restores `$RA (fact(2))` and `$FP (fact(2))`, before popping the stack frame and returning the control to `fact(2)`. All recursive procedure calls of `fact` are wrapped up similarly until, eventually, `fact(10)` returns 10! to main. This is illustrated in Figures 2.18 to 2.22.

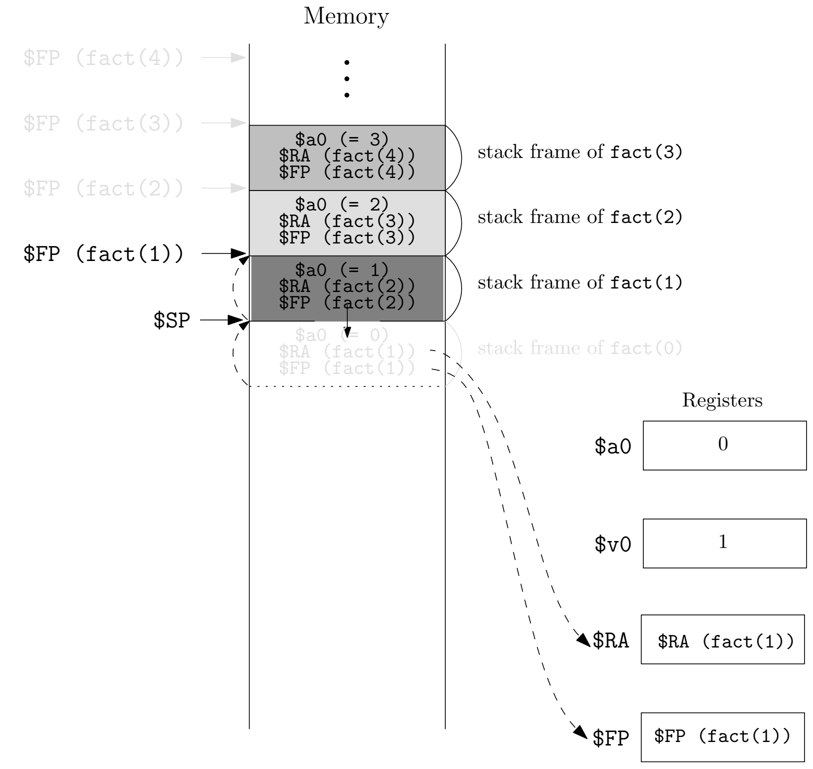


Figure 2.18: Return from fact(0).

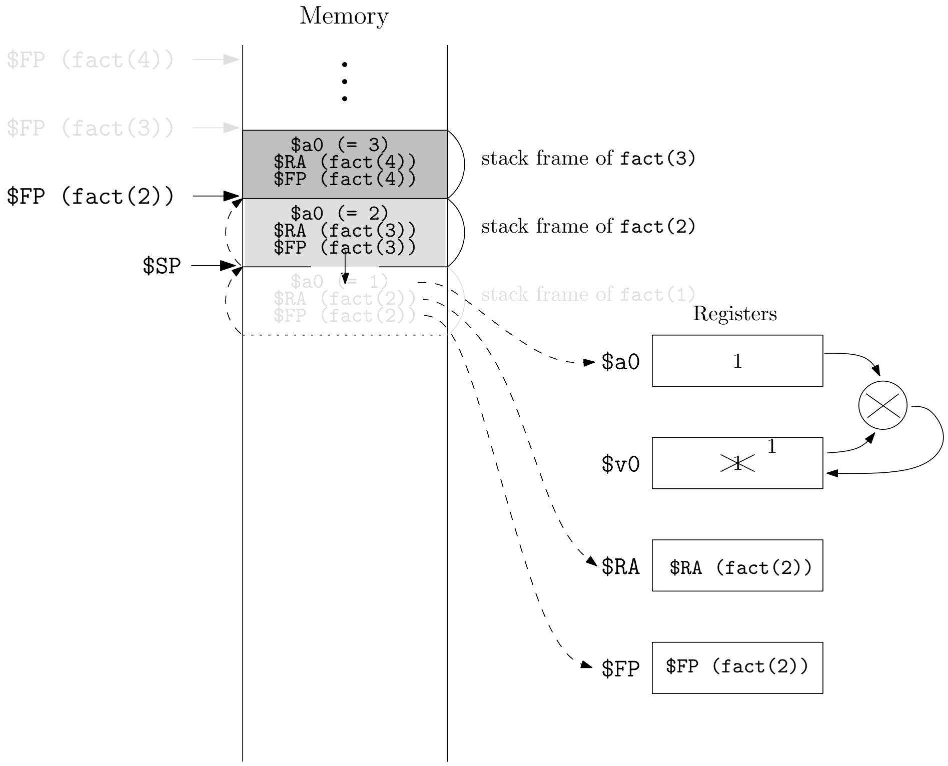


Figure 2.19: Return from fact(1).

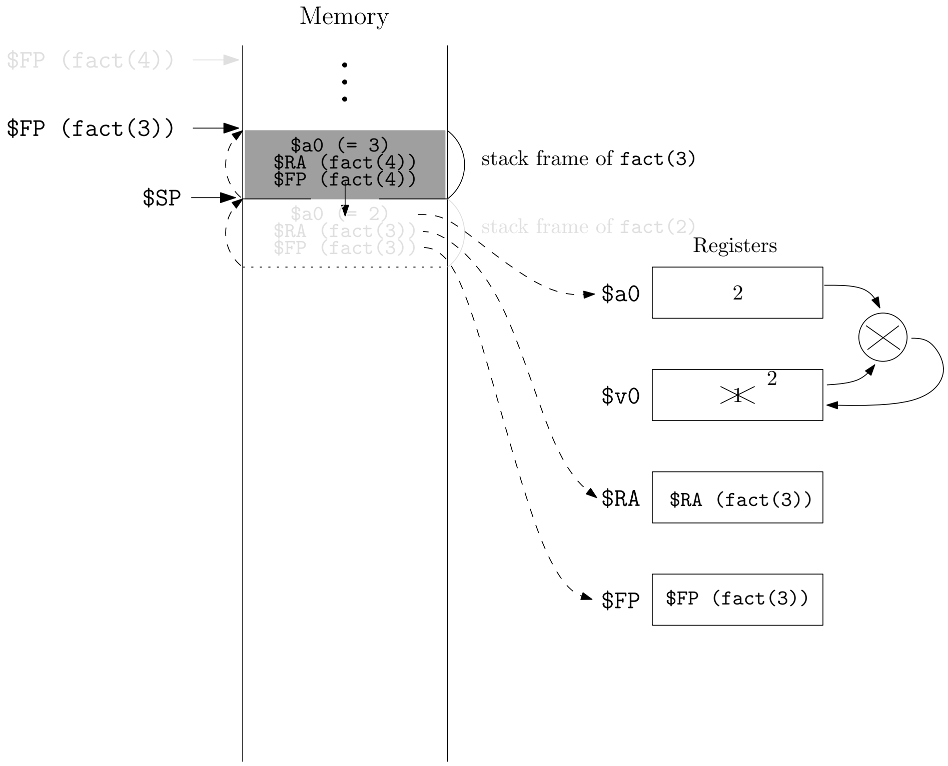


Figure 2.20: Return from fact(2).

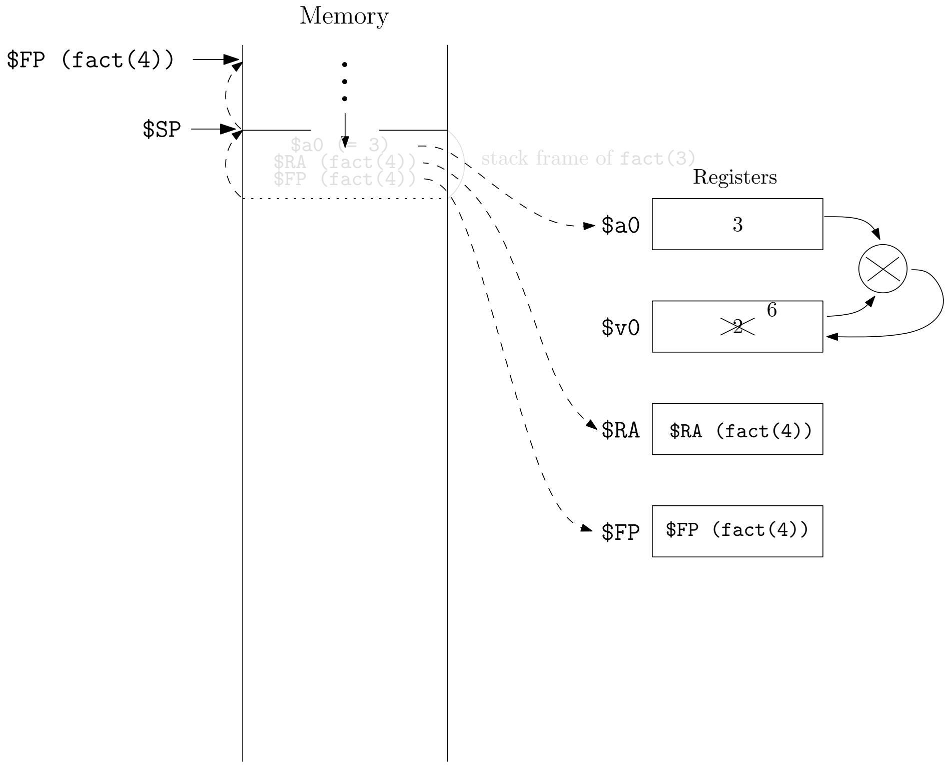


Figure 2.21: Return from fact(3).

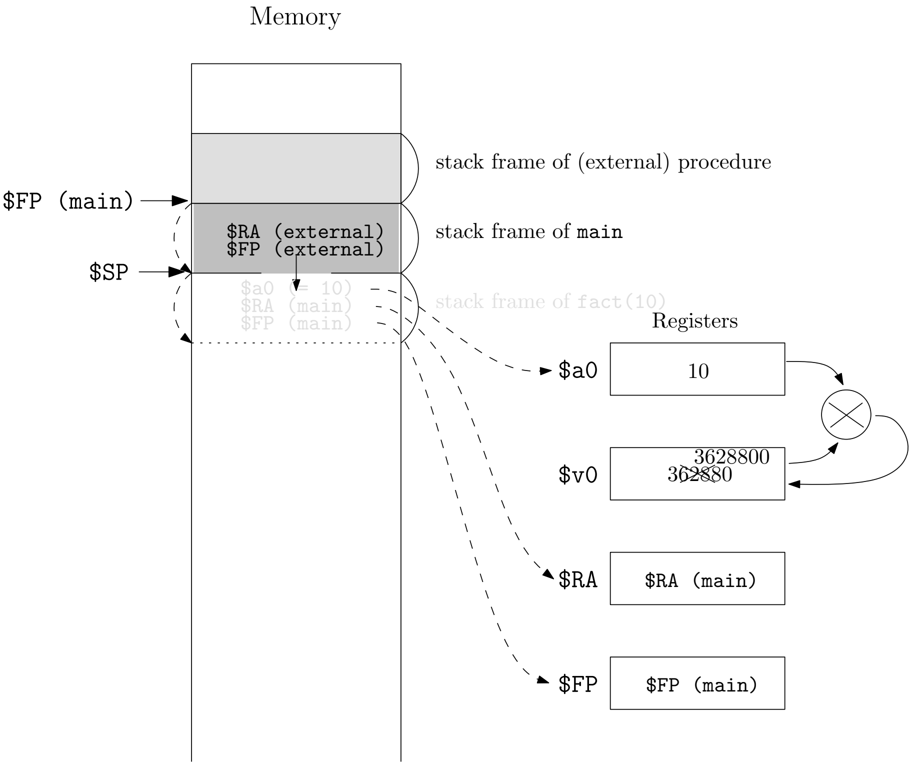


Figure 2.22: Return from fact(10).