# Chapter 2 Processor Architecture
# (Revision number 19)

Two architectural issues surround processor design: the instruction set and the organization of the machine. There was a time, in the early days of computing (circa 60's ad 70's) when processor design was viewed as entirely an exercise in hardware left to electrical engineers. Computers were programmed largely in assembly language and so the fancier the instruction set the simpler the application programs. That was the prevailing conventional wisdom. With the advent of modern programming languages such as Algol in the 60's, and rapid advances in compiler technology, it became very clear that processor design is not simply a hardware exercise. In particular, the instruction-set design is intimately related to how effectively the compiler can generate code for the processor. In this sense, programming languages have exerted a considerable influence on instruction-set design.

Let us understand how the programming language exerts influence on instruction set design. The constructs in high-level language such as assignment statements and expressions map to arithmetic and logic instructions and load/store instructions. High-level languages support data abstractions that may require different precision of operands and addressing modes in the instruction-set. Conditional statements and looping constructs would require conditional and unconditional branching instructions. Further, supporting modularization constructs such as procedures in high-level language may require additional supporting abstractions from the processor architecture.

Applications have a significant influence on the instruction-set design as well. For example, the early days of computing was dominated by scientific and engineering applications. Correspondingly, high-end systems of the 70's and 80's supported floating-point arithmetic in the instruction-set. Circa 2008, the dominant use of computing is in cell phones and other embedded systems, and this trend will no doubt continue as computing weaves its way into the very fabric of society. Streaming applications such as audio and video are becoming commonplace in such gadgets. Correspondingly, requirements of such applications (for e.g., a single instruction operating on a set of data items) are influencing the instruction-set design.

It may not always be feasible or cost-effective to support the need of a particular system software or application directly in hardware. For example, in the early days of computing, low-end computers supported floating-point arithmetic via software libraries that are implemented using integer arithmetic available in the instruction-set. Even to this day, complex operations such as finding the cosine of an angle may not be supported directly by the instruction-set in a general-purpose processor. Instead, special system software called *math libraries* implements such complex operations by mapping them to simpler instructions that are part of the instruction-set.

The operating system has influence on instruction set design as well. A processor may appear to be running several programs at the same time. Think about your PC or PDA.

There are several programs running but there are not several processors. Therefore, there needs to be a way of remembering what a particular program is doing before we go on to another program. You may have seen an efficient cook working on four woks simultaneously making four different dishes. She remembers what state each dish is in and at appropriate times adds the right ingredients to the dishes. The operating system is the software entity (i.e., a program in itself) that orchestrates different program executions on the processor. It has its own influence on processor design as will become apparent in later chapters that deal with program discontinuities and memory management.

## 2.1 What is involved in processor design?

There are hardware resources that we know about from a course on logic design such as registers, arithmetic and logic unit, and the datapath that connects all these resources together. Of course, there are other resources such as main memory for holding programs and data, multiplexers for selecting from a set of input sources, buses for interconnecting the processor resources to the memory, and drivers for putting information from the resources in the datapath onto the buses. We will visit datapath design shortly.

As an analogy, we can think of these hardware resources as the alphabet of a language like English. Words use the alphabet to make up the English language. In a similar manner, the instruction set of the processor uses these hardware resources to give shape and character to a processor. Just as the repertoire of words in a natural language allows us to express different thoughts and emotions, the instruction set allows us to orchestrate the hardware resources to do different things in a processor. Thus, the instruction set is the key element that distinguishes an Intel x86 processor from a Power PC and so on.

As computer users we know we can program a computer at different levels: in languages such as C, Python, and Java; in assembly language; or directly in machine language.

The instruction set is the prescription given by the computer architect as to the capabilities needed in the machine and that should be made visible to the machine language programmer. Therefore, the instruction set serves as a contract between the software (i.e., the programs that run on the computer at whatever level) and the actual hardware implementation. There is a range of choices in terms of implementing the instruction set and we will discuss such choices in later chapters. First, we will explore the issues in designing an instruction set.

## 2.2 How do we design an instruction set?

As we know, computers evolved from calculating machines, and during the early stages of computer design the choice of instructions to have in a machine was largely dictated by whether it was feasible to implement the instruction in hardware. This was because the hardware was very expensive and the programming was done directly in assembly language. Therefore, the design of the instruction set was largely in the purview of the electrical engineers who had a good idea of the implementation feasibilities. However, hardware costs came down and as programming matured, high-level languages were developed such that the attention shifted from implementation feasibility to whether the instructions were actually useful; we mean from the point of producing highly efficient and/or compact code for programs written in such high-level languages.

It turns out that while the instruction set orchestrates what the processor does internally, users of computers seldom have to deal with it directly. Certainly, when you are playing a video game, you are not worried about what instructions the processor is executing when you hit at a target. Anyone who has had an exposure to machine language programming knows how error prone that is.

The shift from people writing assembly language programs to compilers translating high-level language programs to machine code is a primary influence on the evolution of the instruction set architecture. This shift implies that we look to a simple set of instructions that will result in efficient code for high-level language constructs.

It is important to state a note of caution. Elegance of instruction-set is important and the architecture community has invested a substantial intellectual effort in that direction. However, an equally important and perhaps a more over-arching concern is the efficacy of the implementation of the instruction-set. We will revisit this matter in the last section of this chapter.

Each high-level language has its own unique syntactic and semantic flavor. Nevertheless, one can identify a baseline of features that are common across most high-level languages. We will identify such a feature set first. We will use compiling such a feature set as a motivating principle in our discussion and development of an instruction-set for a processor.

## 2.3 A Common High-Level Language Feature Set

Let us consider the following feature set:

1. Expressions and assignment statements: Compiling such constructs will reveal many of the nuances in an instruction-set architecture (ISA for short) from the kinds of arithmetic and logic operations to the size and location of the operands needed in an instruction.
2. High-level data abstractions: Compiling aggregation of simple variables (usually called structures or records in a high-level language) will reveal additional nuances that may be needed in an ISA.
3. Conditional statements and loops: Compiling such constructs result in changing the sequential flow of execution of the program and would need additional machinery in the ISA.
4. Procedure calls: Procedures allow the development of modular and maintainable code. Compiling a procedure call/return brings additional challenges in the design of an ISA since it requires remembering the state of the program before and after executing the procedure, as well as in passing parameters to the procedure and receiving results from the called procedure.

In the next several subsections (Sections 2.4 – 2.8), we will consider each of these features and develop the machinery needed in an ISA from the point of view of efficiently compiling them.

### 2.4 Expressions and Assignment Statements

We know that any high level language (such as Java, C, and Perl) has arithmetic and logical expressions, and assignment statements:

```
a = b + c;  /* add b and c and place in a */          (1)
d = e – f;  /* subtract f  from e and place in d */  (2)
x = y & z;  /* AND y and z and place in x */          (3)
```

Each of the above statements takes *two operands* as inputs, performs an operation on them, and then stores the result in a *third operand*.

Consider the following three instructions in a processor instruction-set:

```
add a, b, c;  a ← b + c           (4)
sub d, e, f;  d ← e - f           (5)
and x, y, z;  x ← y & z           (6)
```

The high level constructs in (1), (2), and (3) directly map to the instructions (4), (5), and (6) respectively.

Such instructions are called *binary* instructions since they work on two operands to produce a result. They are also called *three operand* instructions since there are 3 operands (two source operands and one destination operand). Do we always need 3 operands in such binary instructions? The short answer is no, but we elaborate the answer to this question in the subsequent sections.

### 2.4.1 Where to keep the operands?

Let us discuss the location of the program variables in the above set of equations: *a, b, c, d, e, f, x, y,* and *z*. A simple model of a processor is as shown in Figure 2.1.

We know that inside the processor there is an arithmetic/logic unit or ALU that performs the operations such as ADD, SUB, AND, OR, and so on. We will now discuss where to keep the operands for these instructions. Let us start the discussion with a simple analogy.

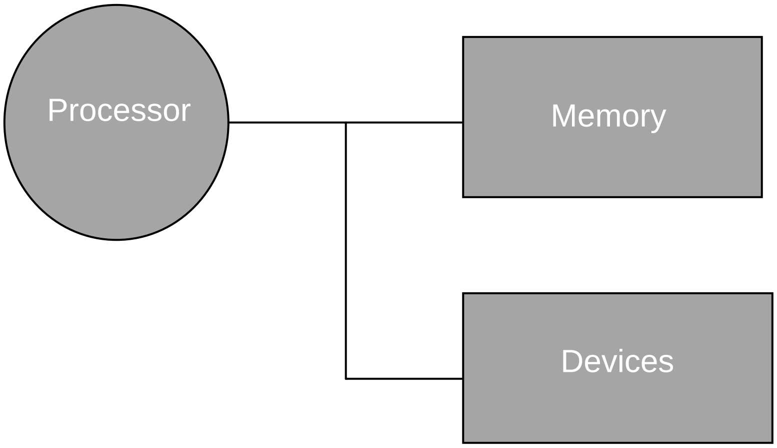


**Figure 2.1: A basic computer organization**

Suppose you have a toolbox that contains a variety of tools. Most toolboxes come with a tool tray. If you are working on a specific project (say fixing a leak in your kitchen faucet), you transfer a set of screwdrivers and pipe wrench from the toolbox into the tool tray. Take the tool tray to the kitchen sink. Work on the leak and then return the tools in the tool tray back into the toolbox when you are all done. Of course, you do not want to run back to the toolbox for each tool you need but instead hope that it is already there in the tool tray. In other words, you are optimizing the number of times you have to run to the toolbox by bringing the minimal set you need in the tool tray.

We want to do exactly that in the design of the instructions as well. We have heard the term *registers* being used in describing the resources available within a processor. These are like memory but they are inside the processor and so they are physically (and therefore electrically) close to the ALU and are made of faster parts than memory. Therefore, if the operands of an instruction are in registers then they are much quicker to access than if they are in memory. But that is not the whole story.

There is another compelling reason, especially with modern processor with very large memory. We refer to this problem as the *addressability* of operands. Let us return to the toolbox/tool-tray analogy. Suppose you are running an automobile repair shop. Now your toolbox is really big. Your project requires pretty much all the tools in your toolbox but at different times as you work on the project. So, as you work on different phases of your project, you will return the current set of tools in your tool tray and bring the appropriate ones for the new phase. Every tool has its unique place of course in the toolbox. On the other hand, you don't have a unique location in the tool tray for each tool. Instead, you are re-using the space in the tool tray for different tools from your toolbox.

An architect faces this same dilemma with respect to uniquely addressing operands. Modern processors have a very large memory system. As the size of memory grows, the size of a memory address (the number of bits needed to name uniquely a memory location) also increases. Thus if an instruction has to name three memory operands, that increases the size of each individual instruction. This is the addressability of operands is a big problem, and will make each instruction occupy several memory locations in order to name uniquely all the memory operands it needs.

On the other hand, by having a small set of registers that cater to the immediate program need (*a la* the tool tray), we can solve this problem since the number of bits to name uniquely a register in this set is small. As a corollary to the addressability problem, the size of the register set has to be necessarily small to limit the number of addressing bits for naming the operands in an instruction (even if the level of integration would permit more registers to be included in the architecture).

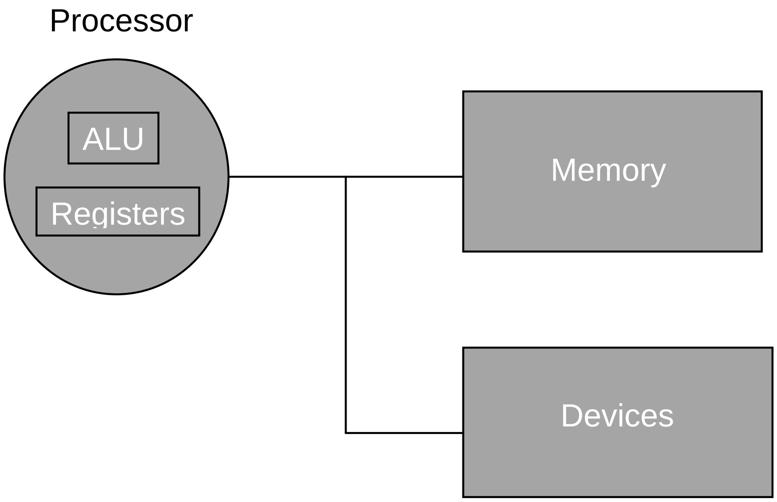


**Figure 2.2: Adding registers inside the processor**

Therefore, we may have instructions that look like:

```
    add r1, r2, r3;  r1 ← r2 + r3
    sub r4, r5, r6;  r4 ← r5 - r6
    and r7, r8, r9;  r7 ← r8 & r9
```

In addition, there is often a need to specify constant values for use in the program. For example, to initialize a register to some starting value or clearing a register that is used as an accumulator. The easiest way to meet this need is to have such constant values be part of the instruction itself. Such values are referred to as *immediate* values.

For example, we may have an instruction that looks like:

```
    addi r1, r2, imm;  r1 ← r2 + imm
```

In this instruction, an immediate value that is part of the instruction becomes the third operand. Immediate values are extremely handy in compiling high-level languages.

**Example 1:**

Given the following instructions

```
ADD   Rx, Ry, Rz    ;      Rx <- Ry + Rz
ADDI  Rx, Ry, Imm   ;      Rx <- Ry + Immediate value
NAND  Rx, Ry, Rz    ;      Rx <-  NOT (Ry AND Rz)
```

Show how you can use the above instructions to achieve the effect of the following instruction

```
SUB Rx, Ry, Rz      ;      Rx <- Ry - Rz
```

**Answer:**

```
NAND  Rz, Rz, Rz    ;      1's complement of Rz in Rz
ADDI  Rz, Rz, 1     ;      2's complement of Rz in Rz
                    ;      Rz now contains -Rz
ADD   Rx, Ry, Rz    ;      Rx <- Ry + (-Rz)
                    ;      Next two instructions restore
                    ;      original value of Rz
NAND  Rz, Rz, Rz    ;      1's complement of Rz in Rz
ADDI  Rz, Rz, 1     ;      2's complement of Rz in Rz
```

All the operands are in registers for these arithmetic and logic operations. We will introduce the concept of ***addressing mode***, which refers to how the operands are specified in an instruction. The addressing mode used in this case called ***register addressing*** since the operands are in registers.

Now with respect to the high-level constructs (1), (2), and (3), we will explore the relationship between the program variables **`a, b, c, d, e, f`** and **`x, y, z`** and these processor registers. As a first order, let us assume that all these program variables reside in memory, placed at well-known locations by the compiler. Since the variables are in memory and the arithmetic/logic instructions work only with registers, they have to be brought into the registers somehow. Therefore, we need additional instructions to move data back and forth between memory and the processor registers. These are called *load* (into registers from memory) and *store* (from registers into memory) instructions.

For example,

```
ld   r2, b;  r2 ← b
st   r1, a;  a  ← r1
```

With these load/store instructions and the arithmetic/logic instructions we can now "compile" a construct such as

```
a = b + c
```

into

```
ld   r2, b                                           (7)
ld   r3, c                                           (8)
add  r1, r2, r3                                      (9)
st   r1, a                                           (10)
```

One may wonder why not simply use memory operands and avoid the use of registers. After all the single instruction

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

seems so elegant and efficient compared to the 4-instruction sequence shown from (7)-(10).

The reason can be best understood with our toolbox/tool tray analogy. You knew you will need to use the screw-driver several times in the course of the project so rather than going to the toolbox every time, you paid the cost of bringing it once in the tool tray and *reuse* it several times before returning it back to the toolbox.

The memory is like the toolbox and the register file is like the tool tray.  You expect that the variables in your program may be used in several expressions.  Consider the following high-level language statement:

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

One can see that once **`a, b,`** and **`c`** are brought from the memory into registers they will be *reused* several times in just this one expression evaluation.  Try compiling the above expression evaluation into a set of instructions (assuming a multiply instruction similar in structure to the add instruction).

In a load instruction, one of the operands is a memory location, and the other is a register that is the destination of the load instruction.  Similarly, in a store instruction the destination is a memory location.

**Example 2:**
An architecture has ONE register called an Accumulator (ACC), and instructions that manipulate memory locations and the ACC as follows:

```
LD     ACC,  a      ;              ACC <- contents of memory location a
ST     a,    ACC    ;              memory location a <- ACC
ADD    ACC,  a      ;              ACC <- ACC + contents of memory location
```

Using the above instructions, show how to realize the semantics of the following instruction:

```
        ADD a, b, c;   memory location a <-
                       contents of memory location b + contents of memory location c
```

**Answer:**

```
        LD      ACC, b
        ADD     ACC, c
        ST      a, ACC
```

### 2.4.2 How do we specify a memory address in an instruction?

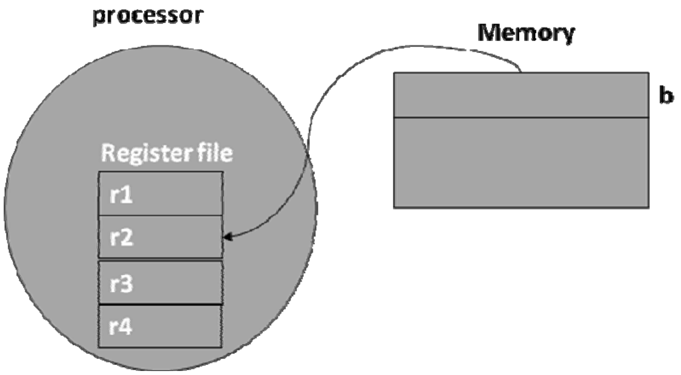


**Figure 2.3: Loading a register from memory**

Let us consider how to specify the memory address as part of the instruction. Of course, we can embed the address in the instruction itself. However, there is a problem with this approach. As we already mentioned earlier in Section 2.4.1, the number of bits needed to represent a memory address in an instruction is already large and will only get worse as the memory size keeps growing. For example, if we have a petabyte (roughly $2^{50}$ bytes) of memory we will need 50 bits for representing each memory operand in an instruction. Further, as we will see later on in Section 2.5, when compiling a program written in a high-level language (especially an object oriented language), the compiler may only know the offset (relative to the base address of the structure) of each member of a complex data structure such as an array or an object. Therefore, we introduce an addressing mode that alleviates the need to have the entire memory address of an operand in each instruction.

Such an addressing mode is ***base+offset*** mode. In this addressing mode, a memory address is computed in the instruction as the sum of the contents of a register in the processor (called a base register) and an offset (contained in the instruction as an immediate value) from that register. This is usually represented mnemonically as

```
    ld   r2, offset(rb);     r2 ← MEMORY[rb + offset]
```

If rb contains the memory address of variable b, and the offset is 0 then the above instruction equivalent to loading the program variable b into the processor register r2.

Note that rb can simply be one of the registers in the processor.

The power of the base+offset addressing mode is it can be used to load/store simple variables as above, and also elements of compound variables (such as arrays and structs) as we will see shortly.

**Example 3:**

Given the following instructions

```
        LW    Rx, Ry, OFFSET      ;       Rx <- MEM[Ry + OFFSET]
        ADD   Rx, Ry, Rz          ;       Rx <- Ry + Rz
        ADDI  Rx, Ry, Imm         ;       Rx <- Ry + Immediate value
```

Show how you can realize a new addressing mode called autoincrement for use with the load instruction that has the following semantics:

```
        LW    Rx, (Ry)+           ;       Rx <- MEM[Ry];
                                  ;       Ry <- Ry + 1;
```

Your answer below should show how the LW instruction using autoincrement will be realized with the given instructions.

**Answer:**

```
        LW    Rx, Ry, 0           ;       Rx <- MEM[Ry + 0]
        ADDI  Ry, Ry, 1           ;       Ry <- Ry + 1
```

### 2.4.3 How wide should each operand be?

This is often referred to as the *granularity* or the *precision* of the operands. To answer this question, we should once again go back to the high-level language and the data types supported in the language. Let's use C as a typical high-level language. The base data types in C are **short, int, long, char**. While the width of these data types are implementation dependent, it is common to have **short** to be 16 bits; **int** to be 32 bits; **char** to be 8 bits. We know that the **char** data type is used to represent alphanumeric characters in C programs. There is a bit of a historic reason why **char** data type is 8 bits. ASCII was introduced as the digital encoding standard for alphanumeric characters (found in typewriters) for information exchange between computers and communication devices. ASCII code uses 7-bits to represent each character. So, one would have expected **char** data type to be 7 bits. However, the popular instruction-set architectures at the time C language was born used 8-bit operands in instructions. Therefore, it was convenient for C to use 8 bits for the **char** data type.

The next issue is the choice of granularity for each operand. This depends on the desired precision for the data type. To optimize on space and time, it is best if the operand size in the instruction matches the precision needed by the data type. This is why processors support multiple precisions in the instruction set: word, half-word, and byte. **Word** precision usually refers to the *maximum precision* the architecture can support in hardware for arithmetic/logic operations. The other precision categories result in less space being occupied in memory for the corresponding data types and hence lead to space efficiency. Since there is less data movement between the memory and processor for such operands there is time efficiency accrued as well.

For the purposes of our discussion, we will assume that a word is 32-bits; half-word is 16-bits; and a byte is 8-bits. They respectively map to *int, short, char* in most C implementations. We already introduced the concept of addressability of operands in Section 2.4.1. When an architecture supports multiple precision operands, there is a question of *addressability* of memory operands. Addressability refers to the smallest precision operand that can be individually addressed in memory. For example, if a machine is byte-addressable then the smallest precision that can be addressed individually is a byte; if it is word addressed then the smallest precision that can be addressed individually is a word. We will assume byte addressability for our discussion. So, a word in memory will look as follows:

| | | | |
|---|---|---|---|
| MSB | | | LSB |

There are 4 bytes in each word (MSB refers to most significant byte; LSB refers to least significant byte). So if we have an integer variable in the program with the value

0x11223344

then its representation in memory it will look as follows:

| | | | |
|---|---|---|---|
| 11 | 22 | 33 | 44 |

Each byte can be individually addressed. Presumably, there are instructions for manipulating at this level of precision in the architecture. So the instruction-set may include instructions at multiple precision levels of operands such as:

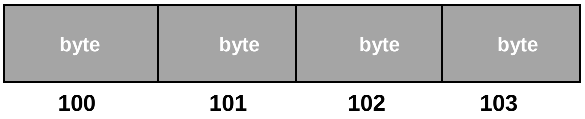
| | |
|---|---|
| **`ld   r1, offset(rb);`** | load a word at address rb+offset into r1 |
| **`ldb  r1, offset(rb);`** | load a byte at address rb+offset into r1 |
| **`add  r1, r2, r3;`** | add word operands in registers r2 and r3 and place the result in r1 |
| **`addb r1, r2, r3;`** | add byte operands in registers r2 and r3 and place the result in r1 |

Necessarily there is a correlation between the architectural decision of supporting multiple precision for the operands and the hardware realization of the processor. The hardware realization includes specifying the width of the datapath and the width of the resources in the datapath such as registers. We discuss processor implementation in much more detail in later chapters (specifically Chapters 3 and 5). Since for our discussion we said word (32 bits) is the maximum precision supported in hardware, it is convenient to assume that the datapath is 32-bits wide. That is, all arithmetic and logic operations take 32-bit operands and manipulate them. Correspondingly, it is convenient to assume that the registers are 32-bits wide to match the datapath width. It should be noted that it is not necessary, but just convenient and more efficient, if the datapath and the register widths match the chosen word width of the architecture. Additionally, the architecture and implementation have to follow some convention for supporting instructions that manipulate precisions smaller than the chosen word width. For example, an instruction such as *addb* that works on 8-bit precision may take the lower 8-bits of the source registers, perform the addition, and place the result in the lower 8-bits of the destination register.

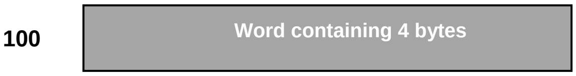
It should be noted that modern architectures have advanced to 64-bit precision for integer arithmetic. Even C programming language has introduced data types with 64-bit precision, although the name of the data type may not be consistent across different implementations of the compiler.

### 2.4.4 Endianness

An interesting question in a byte-addressable machine is the ordering of the bytes within the word. With 4 bytes in each word, if the machine is byte-addressable then four consecutive bytes in memory starting at address 100 will have addresses 100, 101, 102, and 103.

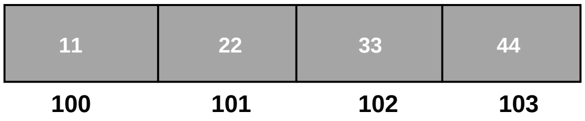


The composite of these 4 bytes make up a word whose address is 100.



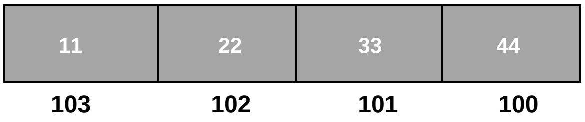
Let us assume the word at location 100 contains the values 0x11223344. The individual bytes within the word may be organized two possible ways:

Organization 1:



In this organization, the MSB of the word (containing the value $11_{hex}$) is at the address of the word operand, namely, 100. This organization is called *big endian.*

Organization 2:



In this organization, the LSB of the word (containing the value $44_{hex}$) is at the address of the word operand, namely, 100. This organization is called *little endian*.

So the endianness of a machine is determined by which byte of the word is at the word address. If it is MSB then it is big endian; if it is LSB then it is little endian.

In principle, from the point of view of programming in high-level languages, this should not matter so long as the program uses the data types in expressions in exactly the same way as they are declared.

However, in a language like C it is possible to use a data type differently from the way it was originally declared.

Consider the following code fragment:

```
int i = 0x11223344;
char *c;

c = (char *) &i;
printf("endian: i = %x; c = %x\n", i, *c);
```

Let us investigate what will be printed as the value of c. This depends on the endianness of the machine. In a big-endian machine, the value printed will be $11_{hex}$; while in a little-endian machine the value printed will be $44_{hex}$. The moral of the story is if you declare a datatype of a particular precision and access it as another precision then it could be a recipe for disaster depending on the endianness of the machine. Architectures such as IBM PowerPC and Sun SPARC are examples of big endian; Intel x86, MIPS, and DEC Alpha are examples of little endian. In general, the endianness of the machine should not have any bearing on program performance, although one could come up with pathological examples where a particular endianness may yield a better performance for a particular program. This particularly occurs during string manipulation.

For example, consider the memory layout of the strings “RAMACHANDRAN” and “WAMACHANDRAN” in a big-endian architecture (see Figure 2.4). Assume that the first string starts at memory location 100.

```
char    a[13] = “RAMACHANDRAN”;
char    b[13] = “WAMACHANDRAN”;
```

|  | +0 | +1 | +2 | +3 |
|---|---|---|---|---|
| 100 | R | A | M | A |
| 104 | C | H | A | N |
| 108 | D | R | A | N |
| 112 | W | A | M | A |
| 116 | C | H | A | N |
| 120 | D | R | A | N |

**Figure 2.4: Big endian layout**

Consider the memory layout of the same strings in a little-endian architecture as shown in Figure 2.5.

| +3 | +2 | +1 | +0 |  |
|---|---|---|---|---|
| A | M | A | R | 100 |
| N | A | H | C | 104 |
| N | A | R | D | 108 |
| A | M | A | W | 112 |
| N | A | H | C | 116 |
| N | A | R | D | 120 |

**Figure 2.5: Little endian layout**

Inspecting Figures 2.4 and 2.5, one can see that the memory layout of the strings is left to right in big-endian while it is right to left in little-endian. If you wanted to compare the strings, one could use the layout of the strings in memory to achieve some coding efficiency in the two architecture styles.

As we mentioned earlier, so long as you manipulate a data type commensurate with its declaration, the endianness of the architecture should not matter to your program. However, there are situations where even if the program does not violate the above rule, endiannesss can come to bite the program behavior. This is particularly true for network code that necessarily cross machine boundaries. If the sending machine is Little-endian and the receiving machine is Big-endian there could even be correctness issues in the resulting network code. It is for this reason, network codes use format conversion routines between host to network format, and vice versa, to avoid such pitfalls[1].

The reader is perhaps wondering why all the box makers couldn't just pick one endianness and go with it? The problem is such things become quasi-religious argument so far as a box maker is concerned, and since there is no standardization the programmers have to just live with the fact that there could be endianness differences in the processors they are dealing with.

To keep the discussion simple, henceforth we will assume a little-endian architecture for the rest of the chapter.

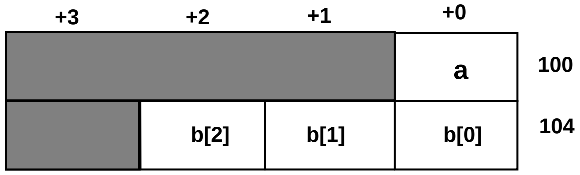
### 2.4.5 Packing of operands and Alignment of word operands

Modern computer systems have large amount of memory. Therefore, it would appear that there is plenty of memory to go around so there should be no reason to be stingy about the use of memory. However, this is not quite true. As memory size is increasing, so is the appetite for memory as far as the applications are concerned. The amount of space occupied a program in memory is often referred to as its *memory footprint*. A compiler, if so directed during compilation, may try **pack** operands of a program in memory to conserve space. This is particularly meaningful if the data structure consists of variables of different granularities (e.g., `int`, `char`, etc.), and if an architecture supports multiple levels of precision of operands. As the name suggests, packing refers to laying out the operands in memory ensuring no wasted space. However, we will also explain in this section why packing may not always be the right approach.

First, let us discuss how the compiler may lay out operands in memory given to conserve space. Consider the data structure

```
struct {
     char a;
     char b[3];
}
```

One possible lay out of this structure in memory starting at location 100 is shown below.



[1] If you have access to Unix source code, look up routines called hton and ntoh, which stand for format conversion between host to network and network to host, respectively.

Let us determine the amount of memory actually needed to layout this data structure. Since each char is 1 byte, the size of the actual data structure is only 4 bytes but the above lay out wastes 50% of the space required to hold the data structure. The shaded region is the wasted space. This is the unpacked layout.

An efficient compiler may eliminate this wasted space and pack the above data structure in memory starting at location 100 as follows:

| | | | |
|---|---|---|---|
| b[2] | b[1] | b[0] | a |
| 103 | 102 | 101 | 100 |

The packing done by the compiler is commensurate with the required precision for the data types and the addressability supported in the architecture. In addition to being frugal with respect to space, one can see that this layout would result in less memory accesses to move the whole structure (consisting of the two variables **a** and **b**) back and forth between the processor registers and memory. Thus, packing operands could result in time efficiency as well.

As we said before, packing may not always be the right strategy for a compiler to adopt.

Consider the following data structure:

```
struct {
     char a;
     int  b;
}
```

We will once again determine how much memory is needed to layout this data structure in memory. A char is 1 byte and an int is 1 word (4 bytes); so totally 5 bytes are needed to store the structure in memory. Let's look at one possible lay out of the structure in memory starting at location 100.

| +3 | +2 | +1 | +0 | |
|---|---|---|---|---|
| b… | b… | $b_{lsb}$ | a | 100 |
| ▨ | ▨ | ▨ | $b_{msb}$ | 104 |

The problem with this layout is that **b** is an int and it starts at address 101 and ends at address 104. To load **b** two words have to be brought from memory (from addresses 100 and 104). This is inefficient whether it is done in hardware or software. Architectures will usually require word operands to start at word addresses. This is usually referred to *alignment restriction* of word operands to word addresses.

An instruction

```
      ld   r2, address
```

will be an illegal instruction if the address is not on a word boundary (100, 104, etc.). While the compiler can generate code to load two words (at addresses 100 and 104) and do the necessary manipulation to get the int data type reconstructed inside the processor, this is inefficient from the point of view of time. So typically, the compiler will layout the data structures such that data types that need word precision are at word address boundaries.

Therefore, a compiler will most likely layout the above structure in memory starting at address 100 as follows:

| +3 | +2 | +1 | +0 | |
|---|---|---|---|---|
| ▨ | ▨ | ▨ | **a** | **100** |
| **b**$_{\mathbf{msb}}$ | **b…** | **b…** | **b**$_{\mathbf{lsb}}$ | **104** |

Note that this layout wastes space (37.5% wasted space) but is more efficient from the point of view of time to access the operands.

You will see this classic space-time tradeoff will surface in computer science at all levels of the abstraction hierarchy presented in Chapter 1 from applications down to the architecture.

### 2.5 High-level data abstractions

Thus far we have discussed simple variables in a high-level language such as *char*, *int*, and *float*. We refer to such variables as *scalars*. The space needed to store such a variable is known *a priori*. A compiler has the option of placing a scalar variable in a register or a memory location. However, when it comes to data abstractions such as arrays and structures that are usually supported in high-level languages, the compiler may have no option except to allocate them in memory. Recall that due to the addressability problem, the register set in a processor is typically only a few tens of registers. Therefore, the sheer size of such data structures precludes allocating them in registers.

### 2.5.1 Structures

Structured data types in a high-level language can be supported with **base+offset** addressing mode.

Consider, the C construct:

```
struct {
     int  a;
     char c;
     int  d;
     long e;
}
```

If the base address of the structure is in some register **`rb`**, then accessing any field within the structure can be accomplished by providing the appropriate offset relative to the base register (the compiler knows how much storage is used for each data type and the alignment of the variables in the memory).

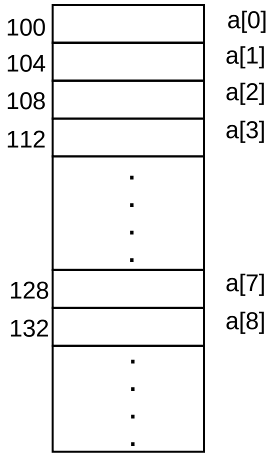
### 2.5.2 Arrays

Consider, the declaration

```
int  a[1000];
```

The name **a** refers not to a single variable but to an array of variables a[0], a[1], etc. For this reason arrays are often referred to as *vectors*. The storage space required for such variables may or may not be known at compile time depending on the semantics of the high-level programming language. Many programming languages allow arrays to be dynamically sized at run time as opposed to compile time. What this means is that at compile time, the compiler may not know the storage requirement of the array. Contrast this with scalars whose storage size requirement is always known to the compiler at compile time. Thus, a complier will typically use memory to allocate space for such vector variables.

The compiler will lay this variable **a** out in memory as follows:



Consider the following statement that manipulates the array:

```
a[7] = a[7] + 1;
```

To compile the above statement, first **`a[7]`** has to be loaded from memory. It is easy to see that this doable given the **base+offset** addressing mode that we already introduced.

```
ld   r1, 28(rb)
```

with **`rb`** initialized to **`100`**, the above instruction accomplishes loading **`a[7]`** into **`r1`**.

Typically, arrays are used in loops. In this case, there may be a loop counter (say ***`j`***) that may be used to index the array. Consider the following statement

```
a[j] = a[j] + 1;
```

In the above statement, the offset to the base register is not fixed. It is derived from the current value of the loop index. While it is possible to still generate code for loading **`a[j]`**[2] this requires additional instructions before the load can be performed to compute the effective address of **`a[7]`**. Therefore, some computer architectures provide an additional addressing mode to let the effective address be computed as the sum of the contents of two registers. This is called the ***base+index*** addressing mode.

Every new instruction and every new addressing mode adds to the complexity of the implementation, thus the pros and cons have to be weighed very carefully. This is usually done by doing a cost/performance analysis. For example, to add base+index addressing mode, we have to ask the following questions:

1. How often will this addressing mode be used during the execution of a program?
2. What is the advantage (in terms of number of instructions saved) by using **base+index** versus **base+offset**?
3. What, if any, is the penalty paid (in terms of additional time for execution) for a load instruction that uses base+index addressing compared to base+offset addressing?
4. What additional hardware is needed to support base+index addressing?

Answers to the above questions, will quantitatively give us a handle on whether or not including the base+index addressing mode is a good idea or not.

We will come back to the issue of how to evaluate adding new instructions and addressing modes to the processor in later chapters when we discuss processor implementation and performance implications.

[2] Basically, the loop index has to be multiplied by 4 and added to rb to get the effective address. Generating the code for loading a[j] using only base+offset addressing mode is left as an exercise to the reader.

## 2.6 Conditional statements and loops

Before we talk about conditional statements, it is important to understand the concept of flow of control during program execution. As we know program execution proceeds sequentially in the normal flow of control:

| | |
|---|---|
| 100 | $I_1$ |
| 104 | $I_2$ |
| 108 | $I_3$ |
| 112 | $I_4$ |
| 116 | $I_5$ |
| 120 | $I_6$ |
| 124 | $I_7$ |
| 128 | $I_8$ |
| 132 | . |
| | . |

Execution of instruction $I_1$ is normally followed by $I_2$, then $I_3$, $I_4$, and so on. We utilize a special register called *Program Counter (PC)*. Conceptually, we can think of the PC pointing to the currently executing instruction[3]. We know that program execution does not follow this sequential path of flow of control all the time.

### 2.6.1 If-then-else statement

Consider,

```
     if (j == k) go to L1;
     a = b + c;
L1:  a = a + 1;
```

Let us investigate what is needed for compiling the above set of statements. The "if" statement has two parts:

1. Evaluation of the predicate "**`j == k`**"; this can be accomplished by using the instructions that we identified already for expression evaluation.
2. If the predicate evaluates to TRUE then it changes the flow of control from going to the next sequential instruction to the target L1. The instructions identified so far do not do accomplish this change of flow of control.

Thus, there is a need for a new instruction that changes the flow of control in addition to evaluating an arithmetic or logic expression. We will introduce a new instruction:

```
beq  r1, r2, L1;
```

[3] Note: We will see in Chapter 3 that for efficacy of implementation, the PC contains the memory address of the instruction immediately following the currently executing instruction.

The semantics of this instruction:

1. Compare r1 and r2
2. If they are equal then the next instruction to be executed is at address L1
3. If they are unequal then the next instruction to be executed is the next one textually following the *beq* instruction.

BEQ is an example of a conditional branch instruction to change the flow of control. We need to specify the address of the target of the branch in the instruction. We know that the PC points to the currently executing instruction. So, the target of a branch can be expressed relative to the current location of the branch instruction by providing an offset as part of the instruction.

```
beq  r1, r2, offset
```

The semantics of this instruction:

1. Compare r1 and r2
2. If they are equal then the next instruction to be executed is at address PC+offset$_{adjusted}$[4]
3. If they are unequal then the next instruction to be executed is the next one textually following the *beq* instruction.

We have essentially added a new addressing mode to computing an effective address. This is called ***PC-relative*** addressing.

The instruction-set architecture may have different flavors of conditional branch instructions such as BNE (branch on not equal), BZ (branch on zero), and BN (branch on negative).

Quite often, there may be an "else" clause in a conditional statement.

```
     if (j == k) {
          a = b + c;
     }
     else {
          a = b – c;
     }
L1:  …..
     …..
```

It turns out that we do not need anything new in the instruction-set architecture to compile the above conditional statement. The conditional branch instruction is sufficient to handle the predicate calculation and branch associated with the "if" statement. However after executing the set of statements in the body of the "if" clause, the flow of control has to be unconditionally transferred to **L1** (the start of statements following the body of the "else" clause).

[4] PC is the address of the beq instruction; offset$_{adjusted}$ is the offset specified in the instruction adjusted for any implementation specifics which we will see in the next Chapter.

To enable this, we will introduce an “unconditional jump” instruction:

```
j     r_target
```

where $\mathbf{r_{target}}$ contains the target address of the unconditional jump.

The need for such an unconditional jump instruction needs some elaboration, since we have a branch instruction already. After all, we could realize the effect of an unconditional branch using the conditional branch instruction **`beq`**. By using the two operands of the **`beq`** instruction to name the same register (e.g., **`beq r1, r1, offset`**), the effect is an unconditional jump. However, there is a catch. The range of the conditional branch instruction is limited to the size of the offset. For example, with an offset size of 8 bits (and assuming the offset is a 2’s complement number so that both positive and negative offsets are possible), the range of the branch is limited to PC-128 and PC+127. This is the reason for introducing a new unconditional jump instruction, wherein a register specifies the target address of the jump.

The reader should feel convinced that using the conditional and unconditional branches that it is possible to compile any level of nested if-then-else statements.

### 2.6.2 Switch statement

Many high level languages provide a special case of conditional statement exemplified by the “switch” statement of C.

```
switch (k) {
     case 0:
     case 1:
     case 2:
     case 3:
     default:
}
```

If the number of cases is limited and/or sparse then it may be best to compile this statement similar to a nested if-then-else construct. On the other hand, if there are a number of contiguous non-sparse cases then compiling the construct as a nested if-then-else will result in inefficient code. Alternatively, using a jump table that holds the starting addresses for the code segments of each of the cases, it is possible to get an efficient implementation. See Figure 2.6.

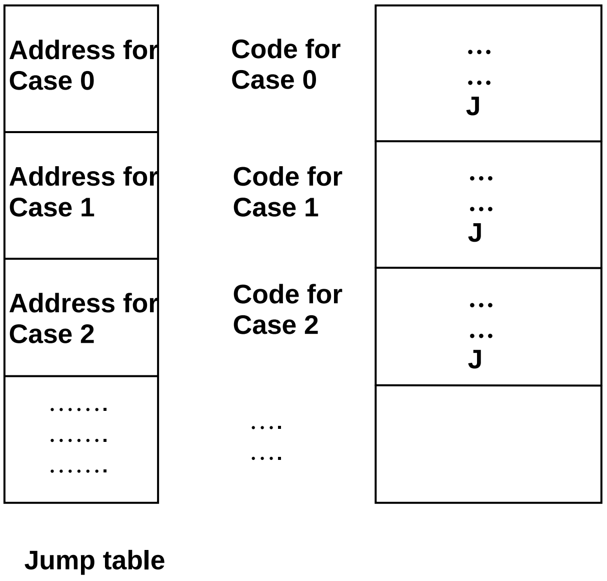


**Figure 2.6: Implementing switch statement with a jump table**

In principle, nothing new is needed architecturally to accomplish this implementation. While nothing new may be needed in the instruction-set architecture to support the switch statement, it may be advantageous to have unconditional jump instruction that works with one level of indirection.

That is,

```
J    @(rtarget)
```

where `rtarget` contains the address of the target address to jump to (unconditionally).

Additionally, some architectures provide special instructions for bounds checking.  For example, MIPS architecture provides a set-on-less-than instruction

```
SLT  s1, s2, s3
```

which works as follows:

```
if s2 < s3 then set s1 to 1
else set s1 to 0
```

This instruction is useful to do the bounds checking in implementing a switch statement.

### 2.6.3 Loop statement

High-level languages provide different flavors of looping constructs.  Consider the code fragment,

```
          j = 0;
loop:     b = b + a[j];
          j = j + 1;
          if (j != 100) go to loop;
```

Assuming a register **`r1`** is used for the variable j, and the value 100 is contained in another register **`r2`**, the above looping construct can be compiled as follows:

```
loop:     …
          …
          …
          bne  r1, r2, loop
```

Thus, no new instruction or addressing mode is needed to support the above looping construct. The reader should feel convinced that any other looping construct (such as “for”, “while”, and “repeat”) can similarly be compiled using conditional and unconditional branch instructions that we have introduced so far.

## 2.7 Checkpoint

So far, we have seen the following high-level language constructs:

1. Expressions and assignment statements
2. High-level data abstractions
3. Conditional statements including loops

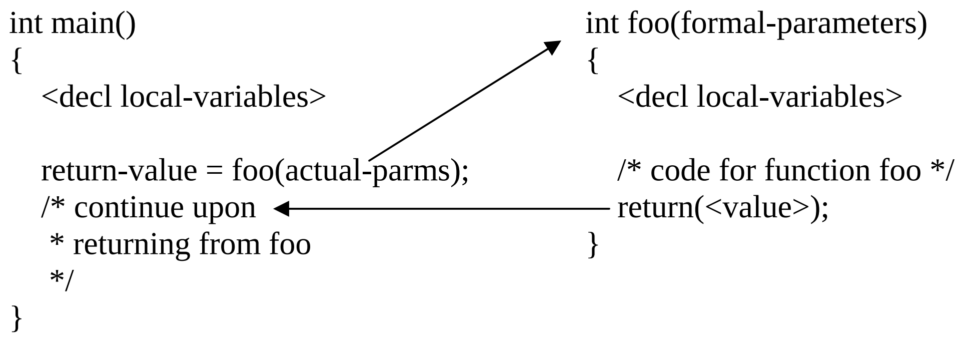
We have developed the following capabilities in the instruction-set architecture to support the efficient compilation of the above constructs:

1. Arithmetic and logic instructions using registers
2. Data movement instructions (load/store) between memory and registers
3. Conditional and unconditional branch instructions
4. Addressing modes: register addressing, base+offset, base+index, PC-relative

## 2.8 Compiling Function calls

Compiling procedures or function calls in a high-level language requires some special attention.

First, let us review the programmer’s mental model of a procedure call. The program is in *main* and makes a call to function *foo*. The flow of control of the program is transferred to the entry point of the function; upon exiting *foo* the flow of control returns to the statement following the call to *foo* in *main*.



```
int main()                                  int foo(formal-parameters)
{                                           {
   <decl local-variables>                      <decl local-variables>

   return-value = foo(actual-parms);           /* code for function foo */
   /* continue upon                            return(<value>);
    * returning from foo                    }
    */
}
```

First, let us define some terminologies: *caller* is the entity that makes the procedure call (*main* in our example); *callee* is the procedure that is being called (*foo* in our example).

Let us enumerate the steps in compiling a procedure call.

1. Ensure that the state of the caller (i.e. processor registers used by the caller) is preserved for resumption upon return from the procedure call
2. Pass the actual parameters to the callee
3. Remember the return address
4. Transfer control to callee
5. Allocate space for callee's local variables
6. Receive the return values from the callee and give them to the caller
7. Return to the point of call

Let us first ask the question whether the above set of requirements can be handled with the capabilities we have already identified in the instruction-set architecture of the processor. Let us look at each of the above items one by one. Let us first understand the ramifications in preserving the state of the caller in Section 2.8.1 and then the remaining chores in Section 2.8.2.

### 2.8.1 State of the Caller

Let us first define what we mean by the state of the caller. To execute the code of the callee, the resources needed are memory (for the code and data of the callee) and processor registers (since all arithmetic/logic instructions use them). The compiler will ensure that the memory used by the caller and callee are distinct (with the exception of any sharing that is mandated by the semantics of the high-level language). Therefore, the contents of the processor registers are the "state" we are worried about since partial results of the caller could be sitting in them at the time of calling the callee. Since we do not know what the callee may do to these registers, it is prudent to *save* them prior to the call and *restore* them upon return.

Now we need some place to save the registers. Let us try a hardware solution to the problem. We will introduce a *shadow register set* into which we will save the processor registers prior to the call; we will restore the processor registers from this shadow register set upon return from the procedure call. See figure 2.7

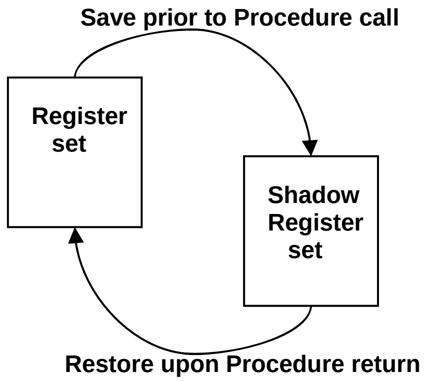


**Figure 2.7: State save/restore for procedure call/return**

We know that procedure calls are frequent in modular codes so rapid save/restore of state is important. Since the shadow register set is located inside the processor, and the save/restore is happening in hardware this seems like a good idea.

The main problem with this idea is that it assumes that the called procedure is not going to make any more procedure calls itself. We know from our experience with high-level languages that this is a bad assumption. In reality, we need as many shadow register sets as the level of nesting that is possible in procedure call sequences. See Figure 2.8.

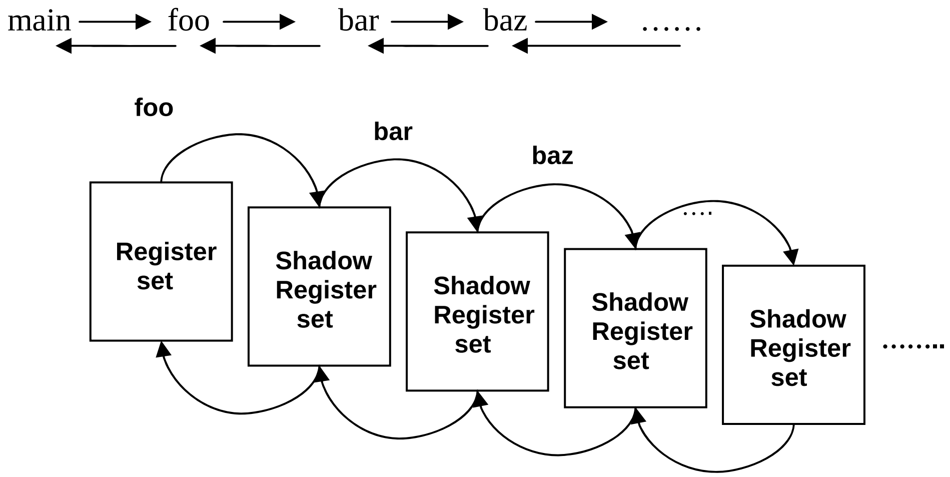


**Figure 2.8: State save/restore for a nested procedure call**

Let us discuss the implications of implementing these shadow register sets in hardware. The level of nesting of procedure calls (i.e., the chain of calls shown in Figure 2.8) is a dynamic property of the program. A hardware solution necessarily has to be finite in terms of the number of such shadow register sets. Besides, this number cannot be arbitrarily large due to the cost and complexity of a hardware solution. Nevertheless, some architectures, such as Sun SPARC, implement a hardware mechanism called *register windowing,* precisely for this purpose. Sun SPARC provides 128 hardware registers, out of which only 32 registers are visible at any point of time. The invisible registers form the shadow register sets to support the procedure calling mechanism shown in Figure 2.8.

Figure 2.8 also suggests a solution that can be implemented in software: the *stack* abstraction that you may have learned in a course on data structures. We save the state in a stack at the point of call and restore it upon return. Since the stack has the *last-in-first-out (LIFO)* property, it fits nicely with the requirement of nested procedure calls.

The stack can be implemented as a software abstraction in memory and thus there is no limitation on the dynamic level of nesting.

The compiler has to maintain a pointer to the stack for saving and restoring state. This does not need anything new from the architecture. The compiler may dedicate one of the processor registers as the *stack pointer*. Note that this is not an architectural restriction but just a convenience for the compiler. Besides, each compiler is free to choose a different register to use as a stack pointer. What this means is that the compiler will not use this register for hosting program variables since the stack pointer serves a dedicated internal function for the compiler.

What we have just done with dedicating a register as the stack pointer is to introduce a software *convention* for register usage.

Well, one of the good points of the hardware solution was the fact that it was done in hardware and hence fast. Now if the stack is implemented in software then it is going to incur the penalty of moving the data back and forth between processor registers and memory on every procedure call/return. Let us see if we can have the flexibility of the software solution while having the performance advantage of the hardware solution. The software solution cannot match the speed advantage of the hardware solution but we can certainly try to reduce the amount of wasted effort.

Let us understand if we do really need to save all the registers at the point of call and restore them upon return. This implicitly assumes that the caller (i.e. the compiler on behalf of the caller) is responsible for save/restore. If the callee does not use ANY of the registers then the whole chore of saving and restoring the registers would have been wasted. Therefore, we could push the chore to the callee and let the callee save/restore registers *it is going to use* in its procedure. Once again, if the caller does not need any of the values in those registers upon return then the effort of the callee is wasted.

To come out of this dilemma, let us carry this idea of software convention a little further. Here again an analogy may help. This is the story of two lazy roommates. They would like to get away with as little house chores as possible. However, they have to eat everyday so they come to an agreement. They have a frying pan for common use and each has a set of their own dishes. The convention they decide to adopt is the following:

- The dishes that are their own they never have to clean.
- If they use the other person’s dishes then they have to leave it clean after every use.

- There is no guarantee that the frying pan will be clean; it is up to each person to clean and use it if they need it.

With this convention, each can get away with very little work…if they do not use the frying pan or the other person's dishes then practically no work at all!

The software convention for procedure calling takes an approach similar to the lazy roommates' analogy. Of course procedure call is not symmetric as the lazy roommates analogy (since there is an ordering - caller to callee). The caller gets a subset of the registers (the **`s`** registers) that are its own. The caller can use them any which it wants and does not have to worry about the callee trashing them. The callee, if it needs to use the **`s`** registers, has to save and restore them. Similar to the frying pan, there is a subset of registers (the **`t`** registers) that are common to both the caller and the callee. Either of them can use the **`t`** registers, and do not have to worry about saving or restoring them. Now, as can be seen, similar to the lazy roommates' analogy, if the caller never needs any values in the **`t`** registers upon return from a procedure it has to do no work on procedure calls. Similarly, if the callee never uses any of the **`s`** registers, then it has to do no work for saving/restoring registers.

The saving/restoring of registers will be done on the stack. We will return to the complete software convention for procedure call/return after we deal with the other items in the list of things to be done for procedure call/return.

### 2.8.2 Remaining chores with procedure calling

1. **Parameter passing:** An expedient way of passing parameters is via processor registers. Once again, the compiler may establish a software convention and reserve some number of processor registers for parameter passing.

   Of course, a procedure may have more parameters than allowed by the convention. In this case, the compiler will pass the additional parameters on the stack. The software convention will establish where exactly on the stack the callee can find the additional parameters with respect to the stack pointer.

2. **Remember the return address:** We introduced the processor resource, Program Counter (PC), early on in the context of branch instructions. None of the high-level constructs we have encountered thus far, have required remembering where we are in the program. So now, there is a need to introduce a new instruction for saving the PC in a well-known place so that it can be used for returning from the callee.

   We introduce a new instruction:

   **JAL** $r_{target}$, $r_{link}$

   The semantics of this instruction is as follows:

   - Remember the return address in $r_{link}$ (which can be any processor register)

- Set PC to the value in $r_{target}$ (the start address of the callee)

We return to the issue of software convention. The compiler may designate one of the processor registers to be $r_{target}$, to hold the address of the target of the subroutine call; and another of the processor registers to be $r_{link}$ to hold the return address. That is these registers are not available to house normal program variables.

So at the point of call, the procedure call is compiled as

```
JAL  r_target, r_link;  /* r_target containing the address
                                  of the callee */
```

Returning from the procedure is straightforward. Since we already have an unconditional jump instruction,

```
J    r_link
```

accomplishes the return from the procedure call.

3. **Transfer control to callee:** Step 3 transfers the control to the callee via the JAL instruction.

4. **Space for callee's local variables:** The stack is a convenient area to allocate the space needed for any local variables in the callee. The software convention will establish where exactly on the stack the callee can find the local variables with respect to the stack pointer.

5. **Return values:** An expedient way for this is for the compiler to reserve some processor registers for the return values. As in the case of parameters, if the number of returned values exceeds the registers reserved by the convention, then the additional return values will be placed on the stack. The software convention will establish where exactly on the stack the caller can find the additional return values with respect to the stack pointer.

6. **Return to the point of call:** As we mentioned earlier, a simple jump through $r_{link}$ will get the control back to the instruction following the point of call.

### 2.8.3 Software Convention

Just to make this discussion concrete, we introduce a set of processor registers and the software convention used:

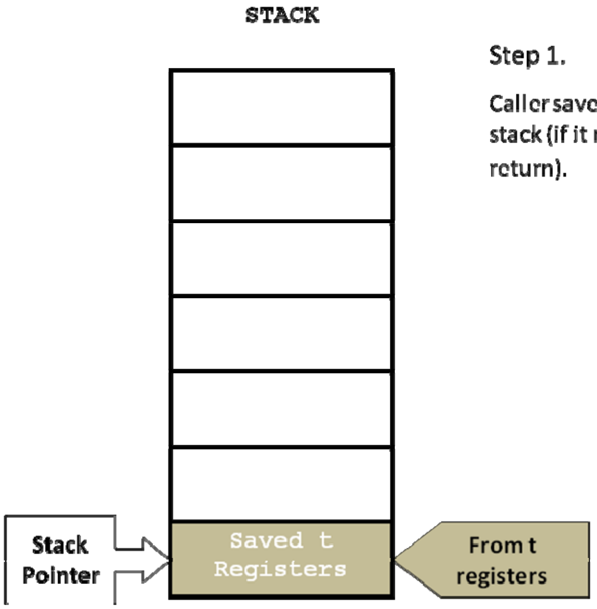
- Registers **`s0-s2`** are the caller's s registers
- Registers **`t0-t2`** are the temporary registers
- Registers **`a0-a2`** are the parameter passing registers
- Register **`v0`** is used for return value

- Register **`ra`** is used for return address
- Register **`at`** is used for target address
- Register **`sp`** is used as a stack pointer

Before illustrating how we will use the above conventions to compile a procedure call we need to recall some details about stacks. A convention that is used by most compilers is that the stack grows down from high addresses to low addresses. The basic stack operations are

- Push: decrements the stack pointer and places the value at the memory location pointed to by the stack pointer
- Pop: takes the value at the memory location pointed to by the stack pointer and increments the stack pointer

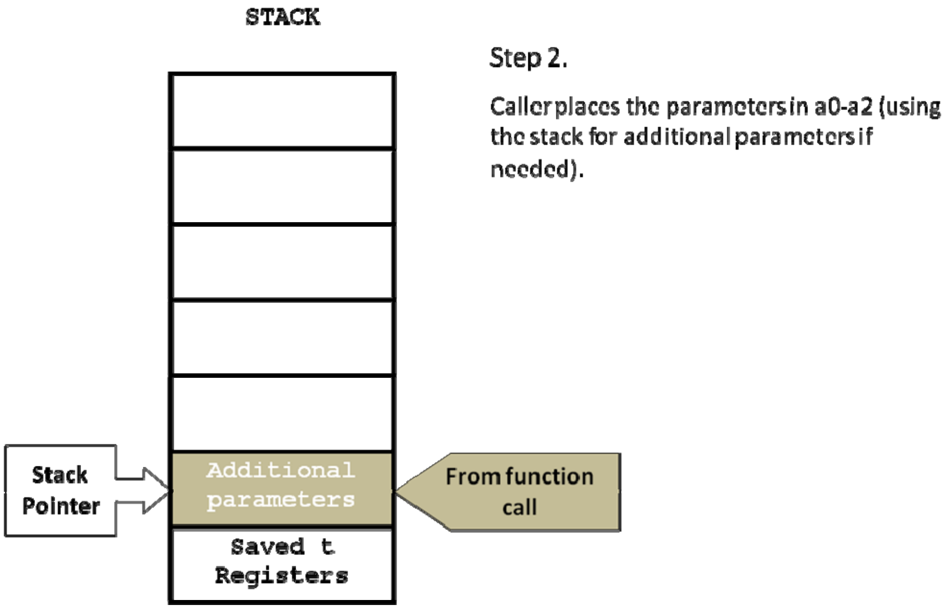
The following illustrations (Figures 2.9-2.20) show the way the compiler will produce code to build the stack frame at run time:



Step 1.

Caller saves any of registers t0-t3 on the stack (if it needs the values in them upon return).

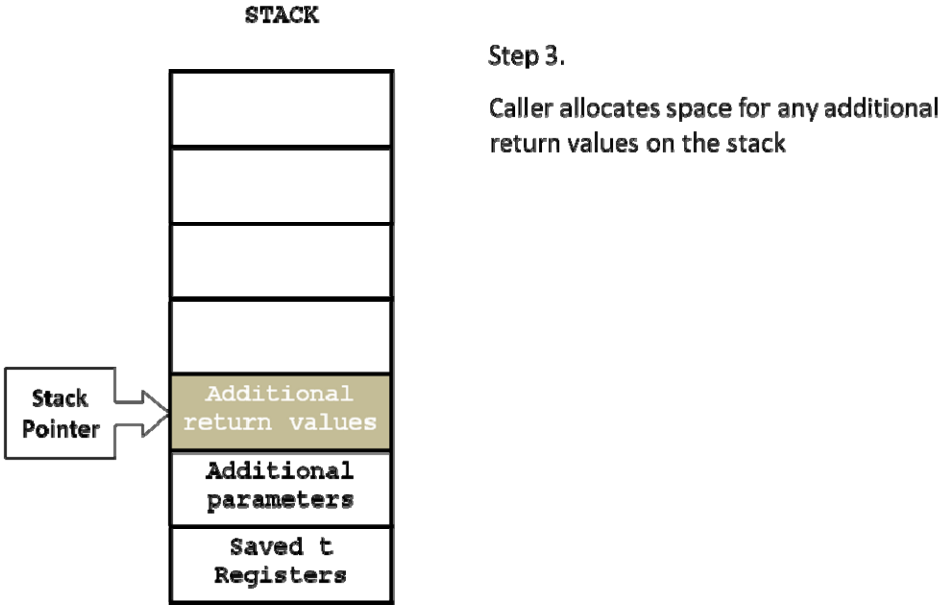
**Figure 2.9: Procedure call/return - Step 1**



Step 2.

Callerplaces the parametersin a0-a2 (using the stack for additional parametersif needed).

Figure 2.10: Procedure call/return - Step 2



Step 3.

Caller allocates space for any additional return values on the stack

Figure 2.11: Procedure call/return - Step 3

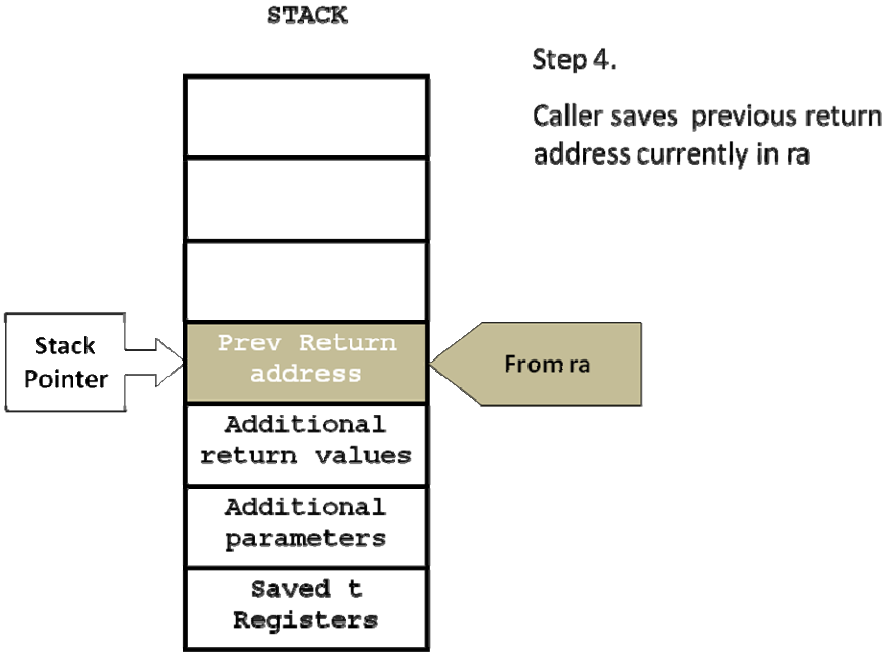


Figure 2.12: Procedure call/return - Step 4

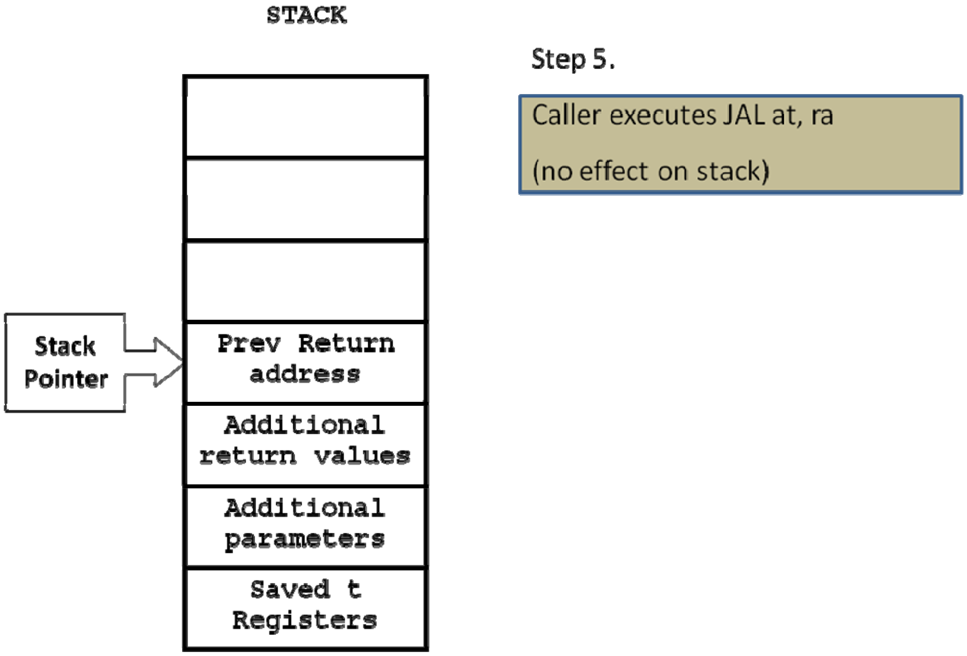


Figure 2.13: Procedure call/return - Step 5

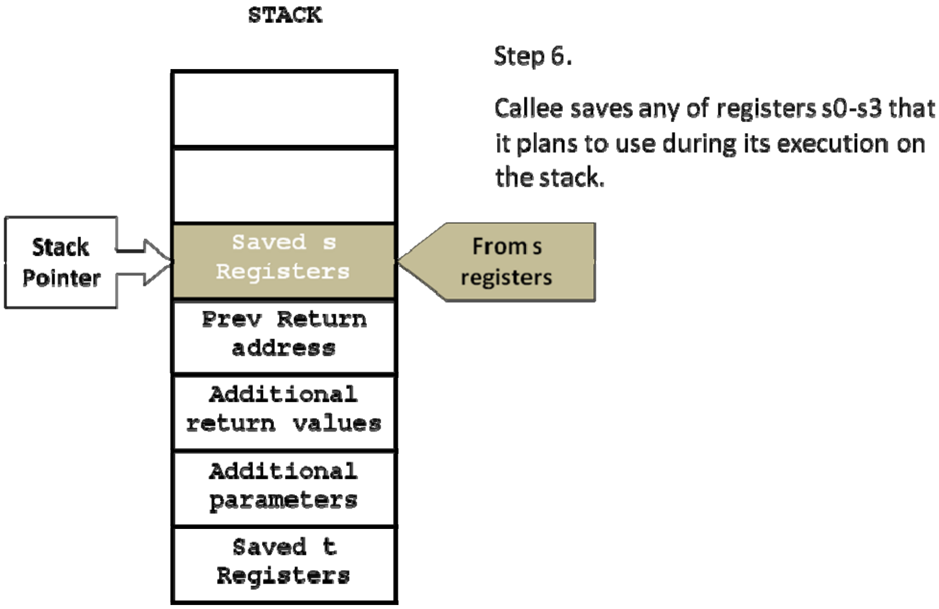


Figure 2.14: Procedure call/return - Step 6

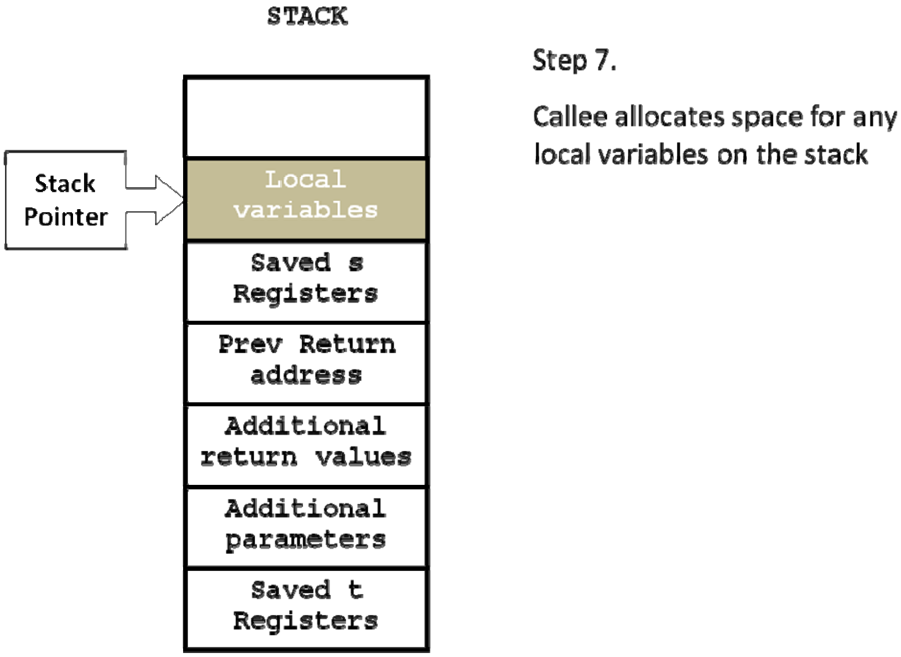


Figure 2.15: Procedure call/return - Step 7

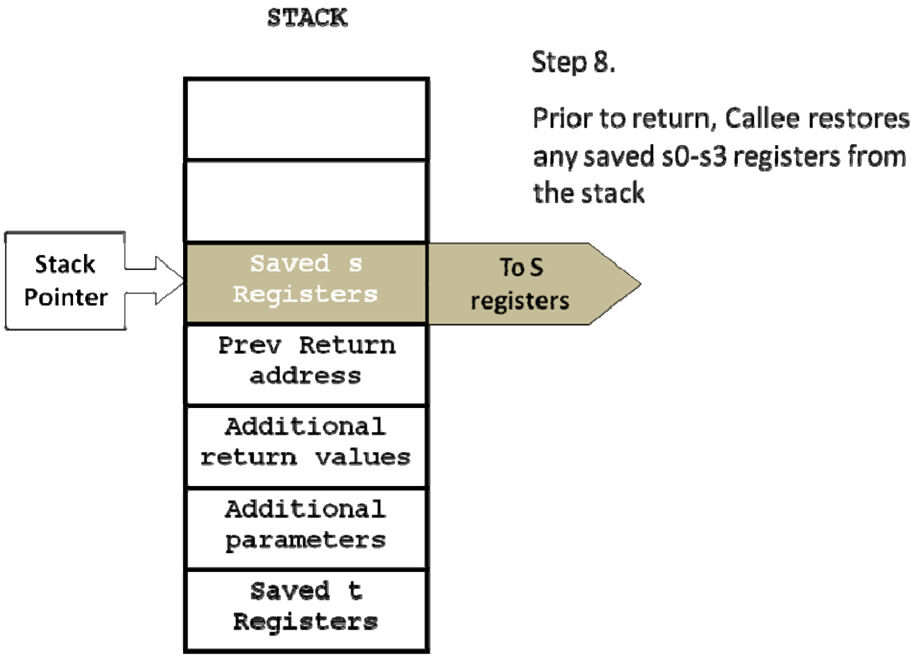


Figure 2.16: Procedure call/return - Step 8

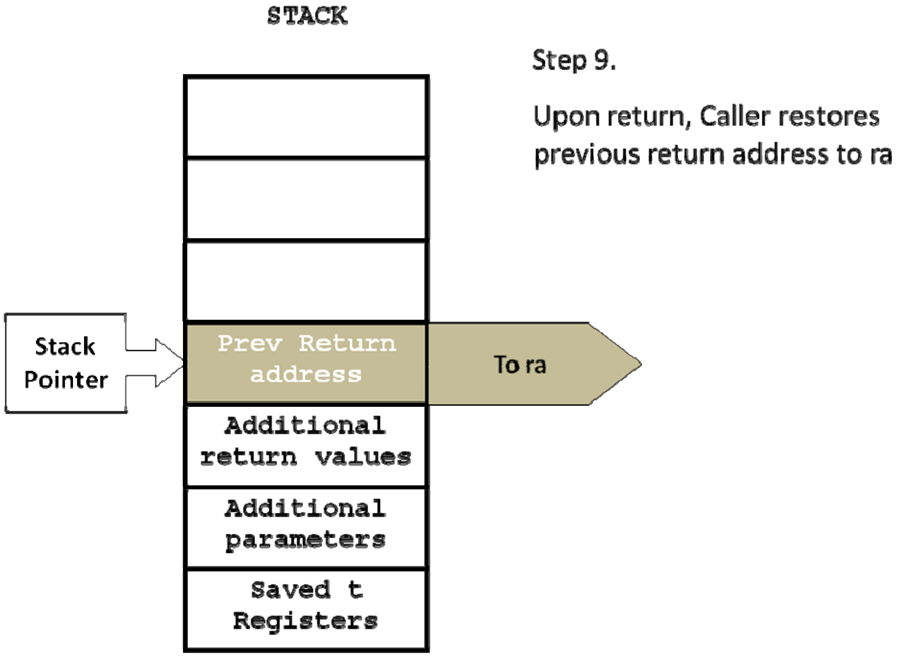


Figure 2.17: Procedure call/return - Step 9

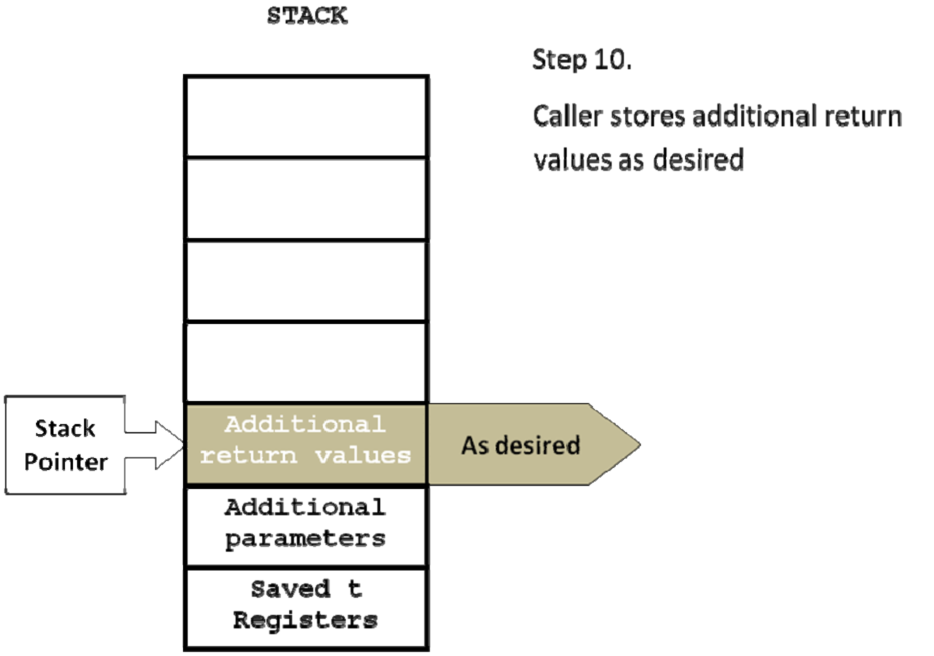


**Figure 2.18: Procedure call/return - Step 10**

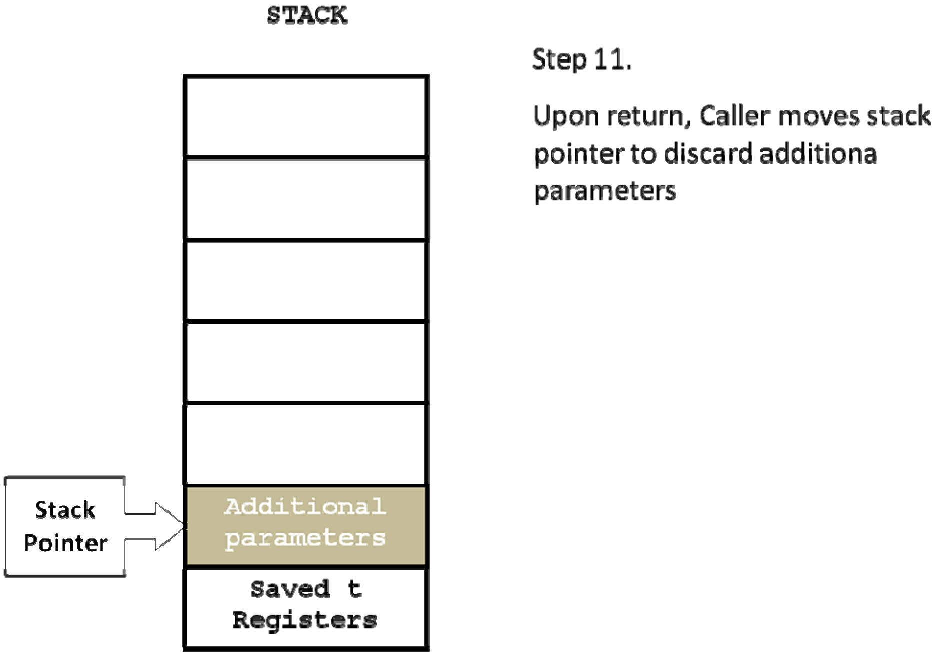

STACK
Step 11.
Upon return, Caller moves stack pointer to discard additiona parameters
Stack Pointer
Additional parameters
Saved t Registers

**Figure 2.19: Procedure call/return - Step 11**

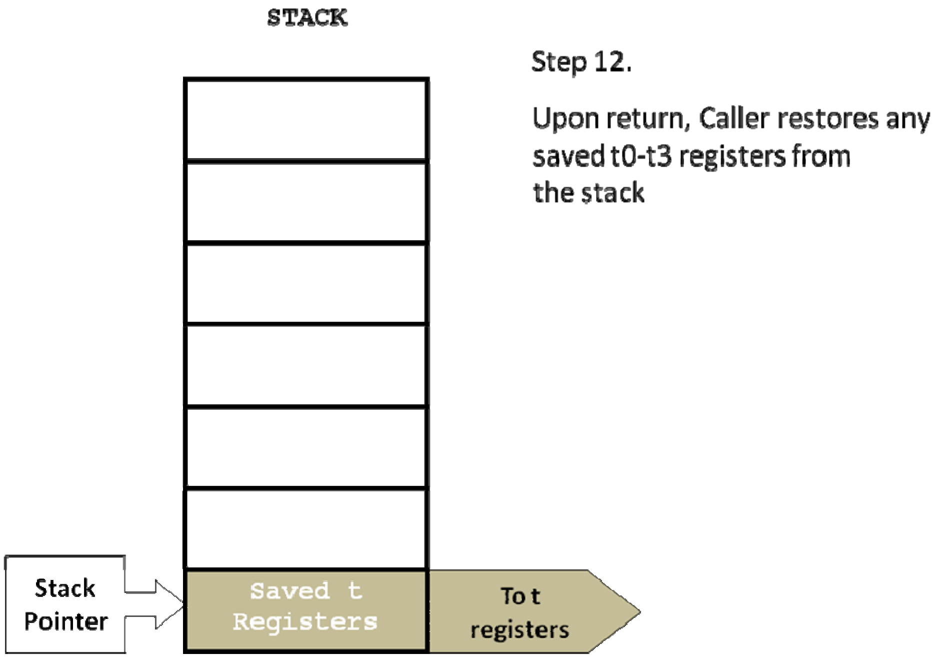

STACK
Step 12.
Upon return, Caller restores any saved t0-t3 registers from the stack
Stack Pointer
Saved t Registers
To t registers

**Figure 2.20: Procedure call/return - Step 12**

### 2.8.4 Activation Record

The portion of the stack that is relevant to the currently executing procedure is called the *activation record* for that procedure. An activation record is the communication area between the caller and the callee. The illustrations in Figure 2.9 to 2.19 show how the activation record is built up by the caller and the callee, used by the callee, and dismantled (by the caller and callee) when control is returned back to the caller. Depending on the nesting of the procedure calls, there could be multiple activation records on the stack. However, at any point of time exactly one activation record is active pertaining to the currently executing procedure.

Consider,

$$\begin{tikzcd}
\text{main} \arrow[r, shift left] & \text{foo} \arrow[l, shift left] \arrow[r, shift left] & \text{bar} \arrow[l, shift left] \arrow[r, shift left] & \text{baz} \arrow[l, shift left] \arrow[r, shift left] & \ldots\ldots \arrow[l, shift left]
\end{tikzcd}$$

Figure 2.21 shows the stack and the activation records for the above sequence of calls.

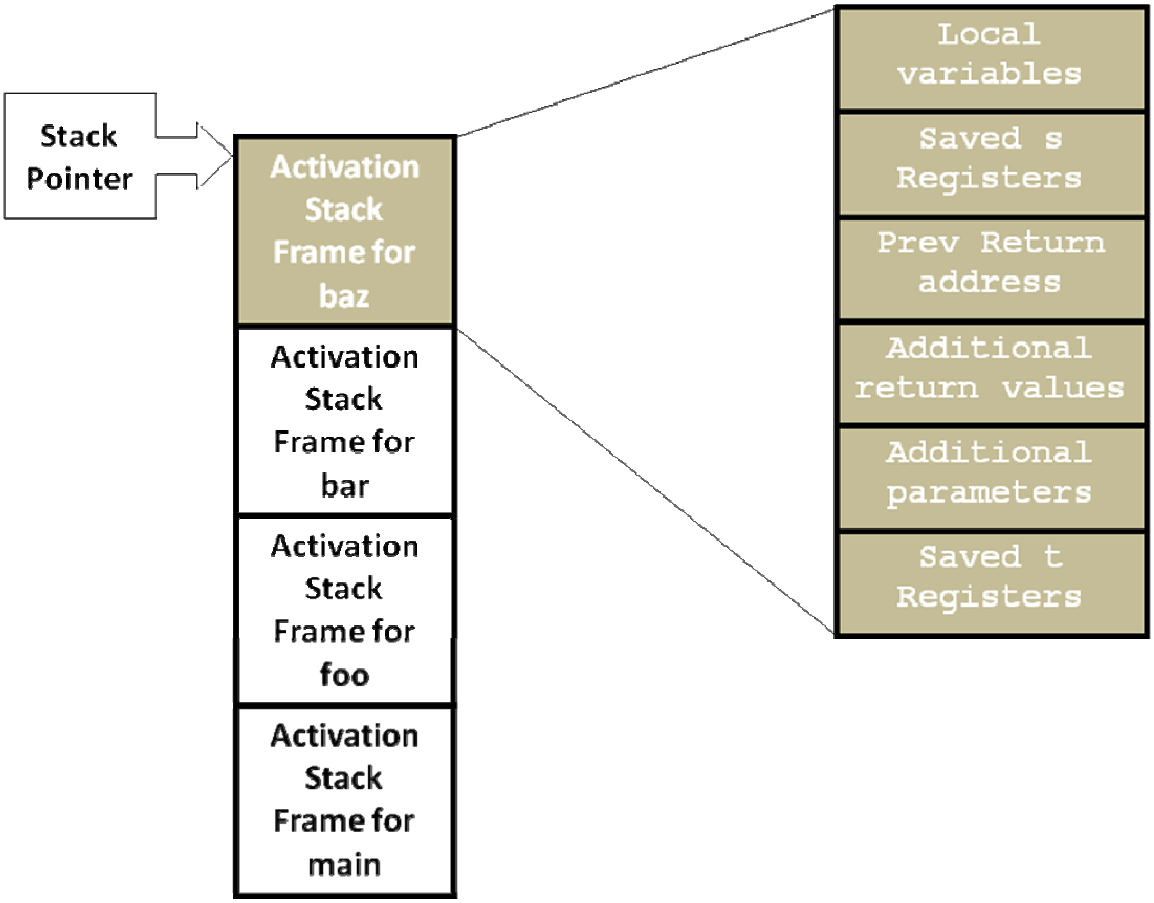


**Figure 2.21: Activation records for a sequence of calls**

### 2.8.5 Recursion

One of the powerful tools to a programmer is recursion. It turns out that we do not need anything special in the instruction-set architecture to support recursion. The stack mechanism ensures that every instantiation of a procedure, whether it is the same procedure or a different one, gets a new activation record. Of course, a procedure that could be called recursively has to be coded in a manner that supports recursion. This is in the purview of the programmer. The instruction-set need not be modified in any way to specially support recursion.

### 2.8.6 Frame Pointer

During execution of a program, it is obviously essential to be able to locate all of the items that are stored on the stack by the program. Their absolute location cannot be known until the program is executing. It might seem obvious that they can be referenced as being at some offset from the stack pointer but there is a problem with this approach. Certain compilers may generate code that will move the stack pointer during the execution of the function (after the stack frame is constructed). For example, a number of languages permit dynamic allocation on the stack. While it would be possible to keep

track of this stack movement the extra bookkeeping and execution time penalties would make this a poor choice. The common solution is to designate one of the general purpose registers as a *frame pointer*. This contains the address of a known point in the activation record for the function and it will never change during execution of the function.
An example will help to understand the problem and the solution. Consider the following procedure:

```
int foo(formal-parameters)
{
     int a, b;

     /* some code */
     if (a > b) {                                   (1)
          int c = 1;                                (2)
          a = a + b + c;                            (3)
     }

     printf(“%d\n, a);                              (4)

     /* more code for foo */

     /*
      * return from foo
      */
     return(0);
}
```

Let us assume that the stack pointer (denoted as $sp) contains 100 after step 6 (Figure 2.14) in a call sequence to foo. In step 7, the callee allocates space for the local variables (a and b). Assuming the usual convention of the stack growing from high addresses to low addresses, a is allocated at address 96 and b is allocated at address 92. Now $sp contains 92.

Procedure foo starts executing. The compiled code for the if statement needs to load a and b into processor registers. This is quite straightforward for the compiler. The following two instructions

```
     ld     r1, 4($sp);     /* load r1 with a */
     ld     r2, 0($sp);     /* load r2 with b */
```

loads a and b into processor registers r1 and r2 respectively.

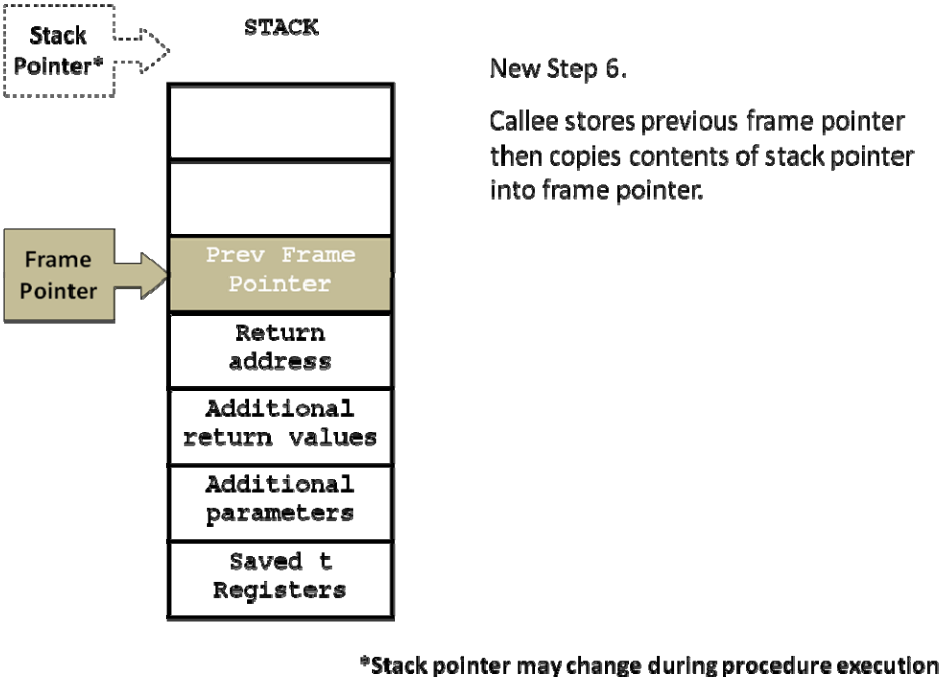
Note what happens next in the procedure if the predicate calculation turns out to be true. The program allocates a new variable c. As one might guess, this variable will also be allocated on the stack. Note however, this is not part of the local variable allocation (step 7, Figure 2.15) prior to beginning the execution of foo. This is a conditional allocation subject to the predicate of the “if” statement evaluating to true. The address of c is 88. Now $sp contains 88.

Inside the code block for the if statement, we may need to load/store the variable *a* and *b* (see statement 3 in the code block). Generating the correct addresses for *a* and *b* is the tricky part. The variable a is used to be at an offset of 4 with respect to $sp. However, it is at an offset of 8 with respect to the current value in $sp.

Once the *if* code block execution completes, *c* is de-allocated from the stack and the $sp changes to 92. Now *a* is at an offset of 4 with respect to the current value in $sp.

The reader can see that from the point of writing the compiler, the fact that the stack can grow and shrink makes the job harder. The offset for local variables on the stack changes depending on the current value of the stack pointer.

This is the reason for dedicating a register as a frame pointer. The frame pointer contains the first address on the stack that pertains to the activation record of the called procedure and never changes while this procedure is in execution. Of course, if a procedure makes another call, then its frame pointer has to be saved and restored by the callee. Thus, the very first thing that the callee does is to save the frame pointer on the stack and copy the current value of the stack pointer into the frame pointer. This is shown in Figure 2.22.



New Step 6.

Callee stores previous frame pointer then copies contents of stack pointer into frame pointer.

*Stack pointer may change during procedure execution

**Figure 2.22: Frame Pointer**

Thus, the frame pointer is a fixed harness on the stack (for a given procedure) and points to the first address of the activation record (AR) of the currently executing procedure.

## 2.9 Instruction-Set Architecture Choices

In this section, we summarize architectural choices in the design of instruction-sets. The choices range from specific set of arithmetic and logic instructions in the repertoire, the addressing modes, architectural style, and the actual memory layout (i.e., format) of the instruction. Sometimes these choices are driven by technology trends at that time and implementation feasibility, while at other times by the goal of elegant and/or efficient support for high-level language constructs.

### 2.9.1 Additional Instructions

Some architectures provide additional instructions to improve the space and time efficiency of compiled code.

- For example, in the MIPS architecture, loads and stores are always for an entire word of 32 bits. Nevertheless, once the instruction is brought into a processor register, special instructions are available for *extracting* a specific byte of the word into another register; similarly, a specific byte of the word can be *inserted* into the word.

- In the DEC Alpha architecture, there are instructions for loading and storing at different operand sizes: byte, half-word, word, and quad-word.

- Some architectures may have some pre-defined immediate values (e.g., 0, 1, and such small integers) available for direct use in instructions.

- DEC VAX architecture has a single instruction to load/store all the registers in the register-file to/from memory. As one might guess, such an instruction would be useful for storing and restoring registers at the point of procedure call and return. The reader should think about the utility of such an instruction in light of the discussion on procedure calling convention that we discussed in this chapter.

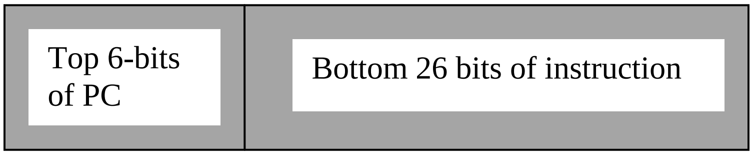
### 2.9.2 Additional addressing modes

In addition to the addressing modes we already discussed, some architectures provide fancier addressing modes.

- Many architectures provide an indirect addressing mode.
    - **`ld @(ra)`**

      In this instruction, the contents of the register ra will be used as the address of the address of the actual memory operand.

- Pseudo-direct addressing
    - MIPS provides this particular addressing mode where the effective address is computed by taking the top 6-bits of the PC and concatenating it with the bottom 26-bits of the instruction word to get a 32-bit absolute address as shown below:



Early architectures such as IBM 360, PDP-11, and VAX 11 supported many more addressing modes than what we covered in this chapter. Most modern architectures have taken a minimalist approach towards addressing modes for accessing memory. This is mainly because over the years it has been seen that very few of the more complex addressing modes are actually used by the compiler. Even base+index is not found in MIPS architecture, though IBM PowerPC and Intel Pentium do support this addressing mode.

### 2.9.3 Architecture styles

Historically, there have been several different kinds of architecture styles.

- **Stack oriented:** Burroughs Computers introduced the stack-oriented architecture where all the operands are on a stack. All instructions work on operands that are on the stack.
- **Memory oriented:** IBM 360 series of machines focused on memory-oriented architectures wherein most (if not all) instructions work on memory operands.
- **Register oriented:** As has been discussed in most of this chapter, most instructions in this architecture deal with operands that are in registers. With the maturation of compiler technology, and the efficient use of registers within the processor, this style of architecture has come to stay as the instruction-set architecture of choice. DEC Alpha, and MIPS are modern day examples of this style of architecture.
- **Hybrid:** As is always the case, one can make a case for each of the above styles of architecture with specific applications or set of applications in mind. So naturally, a combination of these styles is quite popular. Both the IBM PowerPC and the Intel x 86 families of architectures are a hybrid of the memory-oriented and register-oriented styles.

### 2.9.4 Instruction Format

Depending on the instruction repertoire, instructions may be grouped into the following classes:

1. **Zero operand instructions**
   Examples include
    - HALT (halts the processor)
    - NOP (does nothing)

Also, if the architecture is stack-oriented then it only has implicit operands for most instructions (except for pushing and popping values explicitly on the stack). Instructions in such an architecture would look like:

- ADD (pop top two elements of the stack, add them, and push the result back on to the stack)
- PUSH <operand> (pushes the operand on to the stack)
- POP <operand> (pops the top element of the stack into the operand)

2. **One operand instructions**
   Examples include instructions that map to unary operations in high-level languages:
   - INC/DEC <operand> (increments or decrements the specified operand by a constant value)
   - NEG <operand> (2's complement of the operand)
   - NOT <operand> (1's complement of the operand)

   Also, unconditional jump instructions usually have only one operand
   - J <target> (PC <- target)

   Also, some older machines (such as DEC's PDP-8) used 1 implicit operand (called the *accumulator*) and one 1 explicit operand. Instructions in such architecture would look like
   - ADD <operand> ( ACC <- ACC+operand)
   - STORE <operand> ( operand <- ACC)
   - LOAD <operand> (ACC <- operand)

3. **Two operand instructions**
   Examples include instructions that map to binary operations in a high level language. The basic idea is one of the operands is used as both source and destination in a binary operation.
   - ADD R1, R2 ( R1 <- R1 + R2)

   Data movement instructions also fall into this class:
   - MOV R1, R2 (R1 <- R2)

4. **Three operand instructions**
   This is the most general kind and we have seen examples of this throughout this chapter. Examples include:
   - ADD $R_{dst}$, $R_{src1}$, $R_{src2}$ ($R_{dst}$ <- $R_{src1}$+$R_{src2}$)
   - LOAD R, Rb, offset (R <- MEM[Rb + offset]

Instruction format deals with how the instruction is laid out in memory. An architecture may have a mixture of the styles that we discussed earlier, and may consist of various types of instructions that need different number of operands in every instruction.

An instruction typically has the following generic format

| | |
|---|---|
| Opcode | Operand specifiers |

At the point of designing the instruction format, we are getting to the actual implementation. This is because the choice of the instruction format has a bearing on the space and time efficiency of the design. A related choice is the actual *encoding* of the fields within an instruction, which pertains to the semantic meaning associated with each bit pattern for a given field. For example, the bit pattern corresponding to ADD, etc.

Broadly, instruction formats may be grouped under two categories:

- **All instructions are of the same length**
  In this format, all instructions have the same length (for e.g. one memory word). What this means is that the same bit position in an instruction may have a different meaning depending on the instruction.
  - *Pros*:
    - This simplifies the implementation due to the fixed size of the instruction.
    - Interpretation of the fields of the instruction can start (speculatively) as soon as the instruction is available as all instructions have fixed length.
  - *Cons*:
    - Since all instructions do not need all the fields (for example, single operand versus multiple operand instructions), there is potential for wasted space in each instruction.
    - We may need additional glue logic (such as decoders and multiplexers) to apply the fields of the instruction to the datapath elements.
    - The instruction-set designer is limited by the fact that all instructions have to fit within a fixed length (usually one word). The limitation becomes apparent in the size of the immediate operands and address offsets specifiable in the instruction.

  MIPS is an example of an architecture that uses instructions of the same length.

Here are examples of instructions from MIPS:

| ADD | Rs | Rt | Rd | 0 | 0x20 |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

| ADDI | Rs | Rt | Immediate Value |
|---|---|---|---|
| 6 | 5 | 5 | 16 |

| BEQ | Rs | Rt | Offset |
|---|---|---|---|
| 6 | 5 | 5 | 16 |

For example, in the above ADD instruction 5-bit field following Rd serves no purpose in the instruction but is there due to the fixed word length of each instruction.

- **Instructions may be of variable length**
  In this format instructions are of variable length, i.e., an instruction may occupy multiple words.
  - *Pros*:
    - There is no wasted space since an instruction occupies exactly the space required for it.
    - The instruction-set designer is not constrained by limited sizes (for example size of immediate fields).
    - There is an opportunity to choose different sizes and encoding for the opcodes, addressing modes, and operands depending on their observed usage by the compiler.
  - *Cons:*
    - This complicates the implementation since the length of the instruction can be discerned only after decoding the opcode. This leads to a sequential interpretation of the instruction and its operands.

  DEC VAX 11 family and the Intel x86 family are examples of architectures with variable length instructions. In VAX 11, the instructions can vary in size from 1 byte to 53 bytes.

It should be emphasized that our intent here is not to suggest that all such architectural styles and instruction formats covered in this section are feasible today. It is to give the reader an exposure to the variety of choices that have been explored in instruction-set design. For example, there have been commercial offerings in the past of stack-oriented architectures (with zero operand instruction format), and accumulator-based machines (with one operand instruction format). However, such architectures are no longer in the mainstream of general-purpose processors.

## 2.10 LC-2200 Instruction Set

As a concrete example of a simple architecture we define the LC-2200. This is a 32-bit register-oriented little-endian architecture with a fixed length instruction format. There are 16 general-purpose registers as well as a separate program counter (PC) register. All addresses are word addresses. The purpose of introducing this instruction set is three-fold:

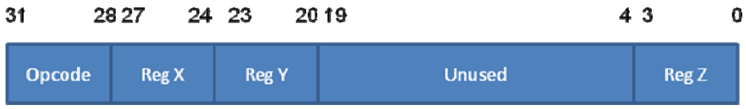
- It serves as a concrete example of a simple instruction-set that can cater to the needs of any high-level language.
- It serves as a concrete architecture to discuss implementation issues in Chapters 3 and 5.
- Perhaps most importantly, it also serves as a simple unencumbered vehicle to add other features to the processor architecture in later chapters when we discuss interrupts, virtual memory and synchronization issues. This is particularly attractive since it exposes the reader to the process by which a particular feature may be added to the processor architecture to serve a specific functionality.

### 2.10.1 Instruction Format

LC-2200 supports four instruction formats. The R-type instruction includes **add** and **nand**. The I-type instruction includes **addi**, **lw**, **sw**, and **beq**. The J-type instruction is **jalr**, and the O-type instruction is **halt**. Thus, totally LC-2200 has only 8 instructions. Table 2.1 summarizes the semantics of these instructions.

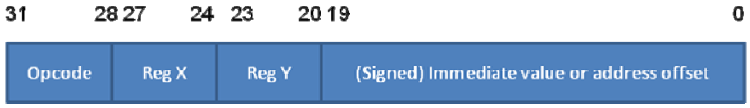
**<u>R-type instructions (add, nand):</u>**

| | |
|---|---|
| bits 31-28: | opcode |
| bits 27-24: | reg X |
| bits 23-20: | reg Y |
| bits 19-4: | unused (should be all 0s) |
| bits 3-0: | reg Z |



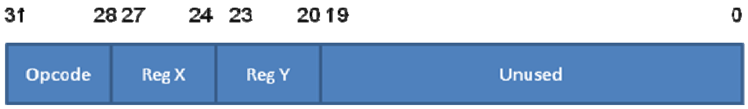
**<u>I-type instructions (addi, lw, sw, beq):</u>**

| | |
|---|---|
| bits 31-28: | opcode |
| bits 27-24: | reg X |
| bits 23-20: | reg Y |
| bits 19-0: | Immediate value or address offset (a 20-bit, 2s complement number with a range of -524288 to +524287) |



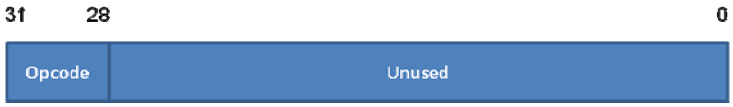
**J-type instructions (jalr):**[5]

bits 31-28: opcode
bits 27-24: reg X (target of the jump)
bits 23-20: reg Y (link register)
bits 19-0: unused (should be all 0s)



**O-type instructions (halt):**

bits 31-28: opcode
bits 27-0: unused (should be all 0s)



[5] LC-2200 does not have a separate unconditional jump instruction. However, it should be easy to see that we can realize such an instruction using JALR $R_{link}$, $R_{dont\text{-}care}$; where $R_{link}$ contains the address to jump to, and $R_{dont\text{-}care}$ is a register whose current value you don't mind trashing.

| **Mnemonic**<br>**Example** | Format | Opcode | Action<br>Register Transfer Language |
|---|---|---|---|
| **add**<br>**add $v0, $a0, $a1** | R | 0<br>0000$_2$ | Add contents of reg Y with contents of reg Z, store results in reg X.<br>RTL: $v0 ← $a0 + $a1 |
| **nand**<br>**nand $v0, $a0, $a1** | R | 1<br>0001$_2$ | Nand contents of reg Y with contents of reg Z, store results in reg X.<br>RTL: $v0 ← ~($a0 && $a1) |
| **addi**<br>**addi $v0, $a0, 25** | I | 2<br>0010$_2$ | Add Immediate value to the contents of reg Y and store the result in reg X.<br>RTL: $v0 ← $a0 + 25 |
| **lw**<br>**lw $v0, 0x42($fp)** | I | 3<br>0011$_2$ | Load reg X from memory. The memory address is formed by adding OFFSET to the contents of reg Y.<br>RTL: $v0 ← MEM[$fp + 0x42] |
| **sw**<br>**sw $a0, 0x42($fp)** | I | 4<br>0100$_2$ | Store reg X into memory. The memory address is formed by adding OFFSET to the contents of reg Y.<br>RTL: MEM[$fp + 0x42] ← $a0 |
| **beq**<br>**beq $a0, $a1, done** | I | 5<br>0101$_2$ | Compare the contents of reg X and reg Y. If they are the same, then branch to the address PC+1+OFFSET, where PC is the address of the beq instruction.<br>RTL: if($a0 == $a1)<br>PC ← PC+1+OFFSET |
| **Note: For programmer convenience (and implementer confusion), the assembler computes the OFFSET value from the number or symbol given in the instruction and the assembler's idea of the PC. In the example, the assembler stores done-(PC+1) in OFFSET so that the machine will branch to label "done" at run time.** | | | |
| **jalr**<br>**jalr $at, $ra** | J | 6<br>0110$_2$ | First store PC+1 into reg Y, where PC is the address of the jalr instruction. Then branch to the address now contained in reg X.<br>Note that if reg X is the same as reg Y, the processor will first store PC+1 into that register, then end up branching to PC+1.<br>RTL: $ra ← PC+1; PC ← $at<br><br>Note that an **unconditional jump** can be realized using **jalr $ra, $t0,** and discarding the value stored in $t0 by the instruction. This is why there is no separate jump instruction in LC-2200. |
| **nop** | n.a. | n.a. | Actually a pseudo instruction (i.e. the assembler will emit: add $zero, $zero, $zero |
| **halt**<br>**halt** | O | 7<br>0111$_2$ | |

**Table 2.1: LC-2200 Instruction Set**

### 2.10.2 LC-2200 Register Set

As we mentioned already, LC-2200 has 16 programmer visible registers. It turns out that zero is a very useful small integer in compiling high-level language programs. For example, it is needed for initializing program variables. For this reason, we dedicate register R0 to always contain the value 0. Writes to R0 are ignored by the architecture.

We give mnemonic names to the 16 registers consistent with the software convention that we introduced in Section 2.8.3. Further, since the assembler needs it, we introduce a '$' sign in front of the mnemonic name for a register. The registers, their mnemonic names, their intended use, and the software convention are summarized in Table 2.2

| **Reg #** | Name | Use | callee-save? |
|---|---|---|---|
| **0** | $zero | always zero (by hardware) | n.a. |
| **1** | $at | reserved for assembler | n.a. |
| **2** | $v0 | return value | No |
| **3** | $a0 | argument | No |
| **4** | $a1 | argument | No |
| **5** | $a2 | argument | No |
| **6** | $t0 | Temporary | No |
| **7** | $t1 | Temporary | No |
| **8** | $t2 | Temporary | No |
| **9** | $s0 | Saved register | YES |
| **10** | $s1 | Saved register | YES |
| **11** | $s2 | Saved register | YES |
| **12** | $k0 | reserved for OS/traps | n.a. |
| **13** | $sp | Stack pointer | No |
| **14** | $fp | Frame pointer | YES |
| **15** | $ra | return address | No |

**Table 2.2: Register convention**

## 2.11 Issues influencing processor design

### 2.11.1 Instruction-set

Throughout this chapter, we focused on instruction-set design. We also made compiling high-level language constructs into efficient machine code as the over-arching concern in the design of an instruction-set. This concern is definitely true up to a point. However, compiler technology and instruction-set design have evolved to a point where this concern is not what is keeping the computer architects awake at nights in companies such as Intel or AMD. In fact, the 80's and 90's saw the emergence of many ISAs, some more elegant than the others, but all driven by the kinds of concerns we articulated in the earlier sections of this chapter. Perhaps one of the favorites from the point of view of elegance and performance is Digital Equipment Corporation's Alpha architecture. The architects of DEC Alpha gave a lot of thought to intuitive and efficient code generation from the point of the compiler writer as well as an ISA design that will lend itself to an efficient implementation. With the demise of DEC, a pioneer in mini computers throughout the 80's and 90's, the Alpha architecture also met its end.

The decade of the 80's saw the debate between *Complex Instruction Set Computers (CISC)* and *Reduced Instruction Set Computers (RISC)*. With CISC-style ISA, the task of a compiler writer is complicated by the plethora of choices that exist for compiling high-level constructs into machine code. Further, the complexity of the ISA makes efficient hardware implementation a significant challenge. Choices for the compiler writer are good in general of course, but if the programmer does not know the implication in terms of performance *a priori*, then such choices become questionable. With the maturation of compiler technology, it was argued that a RISC-style ISA is both easier to use for the compiler writer and would lead to better implementation efficiency than a CISC-style ISA.

As we all know, an ISA that has stood the test of time is Intel's x86. It represents a CISC-style ISA. Yet, x86 is still the dominant ISA while many elegant ones such DEC Alpha have disappeared. The reason is there are too many other factors (market pressure[6] being a primary one) that determine the success or failure of an ISA. Performance is an important consideration of course, but the performance advantage of a really good ISA such as Alpha to that of x86 is not large enough to make that a key determinant. Besides, despite the implementation challenge posed by the x86 ISA, smart architects at Intel and AMD have managed to come up with efficient implementation, which when coupled with continuously increasing clock speeds of processors, make the performance advantage of a really "good" ISA not that significant; at least not significant enough to displace a well entrenched ISA such as x86.

Ultimately, the success or failure of an ISA largely depends on market adoption. Today, computer software powers everything from commerce to entertainment. Therefore, adoption of an ISA by major software vendors (such as Microsoft, Google, IBM, and Apple) is a key factor that determines the success of an ISA. An equally important factor is the adoption of processors embodying the ISA by "box makers" (such as Dell, HP, Apple, and IBM). In addition to the traditional markets (laptops, desktops, and servers), embedded systems (such as game consoles, cell phones, PDAs, and automobiles) have emerged as dominant players in the computing landscape. It is not easy to pinpoint why an ISA may or may not be adopted by each of these segments (software giants, box makers, and builders of embedded systems). While it is tempting to think that such decisions are based solely on the elegance of an ISA, we know from the history of computing that this is not exactly true. Such decisions are often based on pragmatics: availability of good compilers (especially for C) for a given ISA, need to support legacy software, etc.

[6] Part of the reason for this market pressure is the necessity to support legacy code, i.e., software developed on older versions of the same processor. This is called *backward compatibility* of a processor, and contributes to the bloated nature of the Intel x86 ISA.

### 2.11.2 Influence of applications on instruction-set design

Applications have in the past and continue to date, influence the design of instruction-set. In the 70's and perhaps into the 80's, computers were primarily used for number crunching in scientific and engineering applications. Such applications rely heavily on floating-point arithmetic. While high-end computers (such as IBM 370 series and Cray) included such instructions in their ISA, the so-called *mini* computers of that era (such as DEC PDP 11 series) did not include them in their ISA. There were successful companies (e.g., *Floating Point Systems Inc.*) that made attached processors for accelerating floating point arithmetic for such mini computers. Now floating-point instructions are a part of any general-purpose processor. Processors (e.g., StrongARM, ARM) that are used in embedded applications such as cellphones and PDAs may not have such instructions. Instead, they realize the effect of floating-point arithmetic using integer instructions for supporting math libraries.

Another example of applications' influence on the ISA is the MMX instructions from Intel. Applications that process audio, video, and graphics deal with *streaming* data, i.e., continuous data such as a movie or music. Such data would be represented as arrays in the memory. The MMX instructions, first introduced by Intel in 1997 in their Pentium line of processors, aimed at dealing with streaming data efficiently by the CPU. The intuition behind these instructions is pretty straightforward. As the name "stream data" suggests, audio, video and graphics applications require the same operation (such as addition) to be applied to corresponding elements of two or more streams. Therefore, it makes sense to have instructions that mimic this behavior. The MMX instructions originally introduced in the Pentium line and its successors, do precisely that. Each instruction (there are 57 of them grouped into categories such as arithmetic, logical, comparison, conversion, shift, and data transfer) takes two operands (each of which is not a scalar but a vector of elements). For example, an add instruction will add the corresponding elements of the two vectors[7].

A more recent example comes from the gaming industry. Interactive video games have become very sophisticated. The graphics and animation processing required to be done in real-time in such gaming consoles have gotten to the point where it is beyond the capability of the general-purpose processors. Of course, you wouldn't want to lug around a supercomputer on your next vacation trip to play video games! Enter *Graphic Processing Units* (*GPU*s for short). These are special-purpose attached processors that perform the needed arithmetic for such gaming consoles. Basically, the GPU comprises a number of functional units (implementing primitive operations required in graphics rendering applications) that operate in parallel on a stream of data. A recent partnership between Sony, IBM, and Toshiba unveiled the Cell processor that takes the concept of GPUs a step further. The Cell processor comprises several processing elements on a

[7] As a historical note, the MMX instructions evolved from a style of parallel architectures called Single Instruction Multiple Data (SIMD for short), which was prevalent until the mid 90's for catering to the needs of image processing applications. See Chapter 11 for an introduction to the different parallel architecture styles.

single chip, each of which can be programmed to do some specialized task. The Cell processor architecture has made its way into PlayStation (PS-3).

### 2.11.3 Other issues driving processor design

ISA is only one design issue, and perhaps not the most riveting one, in the design of modern processors. We will list some of the more demanding issues in this section, some of which will be elaborated in later chapters.

1. **Operating system:** We already mentioned that operating system plays a crucial role in the processor design. One manifestation of that role is giving to the programmer an illusion of memory space that is larger than the actual amount of memory that is present in the system. Another manifestation is the responsiveness of the processor to external events such as interrupts. As we will see in later chapters, to efficiently support such requirements stemming from the operating system, the processor may include new instructions as well as new architectural mechanisms that are not necessarily visible via the ISA.
2. **Support for modern languages:** Most modern languages such as Java, C++, and C# provide the programmer with the ability to dynamically grow and shrink the data size of the program. Dubbed *dynamic memory allocation*, this is a powerful feature both from the point of view of the application programmer and from the point of view of resource management by the operating system. Reclaiming memory when the data size shrinks, referred to as *garbage collection* is crucial from the point of view of resource management. Mechanisms in the processor architecture for automatic garbage collection are another modern day concern in processor design.
3. **Memory system:** As you very well know, processor speeds have been appreciating exponentially over the past decade. For example, a Sun 3/50 had a processor speed of 0.5 MHz circa 1986. Today circa 2007, laptops and desktops have processor speeds in excess of 2 GHz. Memory density has been increasing at an exponential rate but memory speed has not increased at the same rate as processor speed. Often this disparity between processor and memory speeds is referred to as the *memory wall*. Clever techniques in the processor design to overcome the memory wall are one of the most important issues in processor design. For example, the design of cache memories and integrating them into the processor design is one such technique. We will cover these issues in a later chapter on memory hierarchies.
4. **Parallelism:** With increasing density of chips usually measured in millions of transistors on a single piece of silicon, it is becoming possible to pack more and more functionality into a single processor. In fact, the chip density has reached a level where it is possible to pack multiple processors on the same piece of silicon. These architectures, called *multi-core* and *many-cores* bring a whole new set of processor design issues[8]. Some of these issues, such as parallel programming, and memory consistency, are carried over from traditional multiprocessors (a box comprising several processors), which we discuss in a later chapter.

[8] Architecturally, there is not a major difference between multi- and many-cores. However, the programming paradigm needs a radical rethinking if there are more than a few (up to 8 or 16) cores. Hence the distinction: multi-core may have up to 8 or 16 cores; anything beyond that is a many-core architecture.

5. **Debugging:** Programs have become complex. An application such as a web server, in addition to being parallel and having a large memory footprint, may also have program components that reach into the network and databases. Naturally, developing such applications is non-trivial. A significant concern in the design of modern processors is support for efficient debugging, especially for parallel program.
6. **Virtualization:** As the complexity of applications increases, their needs have become complex as well. For example, an application may need some services available only in one particular operating system. If we want to run multiple applications simultaneously, each having its own unique requirements, then there may be a need to simultaneously support multiple application execution environments. You may have a dual boot laptop, and your own unique reason for having this capability. It would be nice if you can have multiple operating systems co-existing simultaneously without you having to switch back and forth. *Virtualization* is the system concept for supporting multiple distinct execution environments in the same computer system. Architects are now paying attention to efficiently supporting this concept in modern processor design.
7. **Fault tolerance:** As the hardware architecture becomes more complex with multi- and many-cores and large memory hierarchies, the probability of component failures also increases. Architects are increasingly paying attention to processor design techniques that will hide such failures from the programmer.
8. **Security:** Computer security is a big issue in this day and age. We normally think of network attacks when we are worried about protecting the security of our computer. It turns out that the security can be violated even within a box (between the memory system and the CPU). Architects are increasingly paying attention to alleviate such concerns by incorporating encryption techniques for processor-memory communication.

## 2.12 Summary

Instruction-set serves as a contract between hardware and software. In this chapter, we started from basics to understand the issues in the design of an instruction set. The important points to take away from the chapter are summarized below:

- Influence of high-level language constructs in shaping the ISA
- Minimal support needed in the ISA for compiling arithmetic and logic expressions, conditional statements, loops, and procedure calls
- Pragmatic issues (such as addressing and access times) that necessitate use of registers in the ISA
- Addressing modes for accessing memory operands in the ISA commensurate with the needs of efficient compilation of high level language constructs
- Software conventions that guide the use of the limited register set available within the processor
- The concept of a software stack and its use in compiling procedure calls
- Possible extensions to a minimal ISA
- Other important issues guiding processor design in this day and age.

## 2.13 Review Questions

1. Having a large register-file is detrimental to the performance of a processor since it results in a large overhead for procedure call/return in high-level languages. Do you agree or disagree? Give supporting arguments.

2. Distinguish between the frame pointer and the stack pointer.

3. In the LC-2200 architecture, where are operands normally found for an add instruction?

4. Endianness: Let's say you want to write a program for comparing two strings. You have a choice of using a 32-bit byte-addressable Big-endian or Little-endian architecture to do this. In either case, you can pack 4 characters in each word of 32-bits. Which one would you choose and how will you write such a program? [Hint: Normally, you would do string comparison one character at a time. If you can do it a word at a time instead of a character at a time, that implementation will be faster.]

5. ISA may support different flavors of conditional branch instructions such as BZ (branch on Zero), BN (branch on negative), and BEQ (branch on equal). Figure out the predicate expressions in an "if" statement that may be best served by these different flavors of conditional branch instructions. Give examples of such predicates in "if" statement, and how you will compile them using these different flavors of branch instructions.

6. We said that endianness will not affect your program performance or correctness so long as the use of a (high level) data structure is commensurate with its declaration. Are there situations where even if your program does not violate the above rule, you could be bitten by the endianness of the architecture? [Hint: Think of programs that cross network boundaries.]

7. Work out the details of the implementing the switch statement using jump tables in assembly using any flavor of conditional branch instruction. [Hint: After ensuring that the value of the switch variable is within the bounds of valid case values, jump to the start of the appropriate code segment corresponding to the current switch value, execute the code and finally jump to exit.]

8. Procedure A has important data in both S and T registers and is about to call procedure B. Which registers should A store on the stack? Which registers should B store on the stack?

9. Is it allowable to modify data stored in the interior of a stack? In other words, are all accesses to data in a stack made via pushes and pops?

10. Is a frame pointer necessary if procedure calls will be used?

11. DEC VAX has a single instruction for loading and storing all the program visible registers from/to memory. Can you see a reason for such an instruction pair? Consider both the pros and cons.

12. Extend the LC-2200 assembly language to include a subtract operation. Give some pseudo code for the actions that must be taken in each state.

13. Is it necessary for the LC series processor instruction set to have a subtract operation, or can subtraction be implemented using existing LC-2200 instructions?

14. Suppose you are writing an LC-2200 program and you want to jump a distance that is farther than allowed by a BEQ instruction. Is there a way to jump to an address?

15. Explain what is meant by an instruction set.

16. What are the influences on instruction set design?

17. What are conditional statements and how are they handled in the ISA?
18. Define the term addressing mode.

19. Where are local (auto) variables normally allocated space?

20. Would you consider the stack that is used for facilitating procedure call/return a hardware abstraction, software abstraction, pass by name calling convention, virtual machine or a queue?

21. Given the following instructions

```
BEQ   Rx, Ry, offset; if (Rx == Ry) PC=PC+offset
SUB    Rx, Ry, Rz      ;          Rx <- Ry - Rz
ADDI  Rx, Ry, Imm   ;          Rx <- Ry + Immediate value
AND   Rx, Ry, Rz      ;          Rx <- Ry AND Rz
```

    Show how you can realize the effect of the following instruction:

```
            BGT Rx, Ry, offset;   if (Rx > Ry) PC=PC+offset
```

    Assume that the registers and the Imm field are 8-bits wide.
    You can ignore the case that the SUB instruction causes an overflow.

22. Given the following load instruction

```
LW      Rx, Ry, OFFSET        ;          Rx <- MEM[Ry + OFFSET]
```

    Show how to realize a new addressing mode, called *indirect*, for use with the load instruction that is represented in assembly language as:

```
LW      Rx, @(Ry)      ;
```

    The semantics of this instruction is that the contents of register Ry is the address of a pointer to the memory operand that must be loaded in Rx.

23. Consider the following program and assume that for this processor:

- All arguments are passed on the stack.
- Register V0 is for return values.
- The S registers are expected to be saved, that is a calling routine can leave values in the S registers and expect it to be there after a call.
- The T registers are expected to be temporary, that is a calling routine must not expect values in the T registers to be preserved after a call.

```
int bar(int a, int b)
{
    /* Code that uses registers T5, T6, S11-S13;  */
    return(1);
}

int foo(int a, int b, int c, int d, int e)
{
    int x, y;
    /* Code that uses registers T5-T10, S11-S13; */
    bar(x, y);  /* call bar */
    /* Code that reuses register T6 and arguments a, b, and c;
    return(0);
}

main(int argc, char **argv)
{
    int p, q, r, s, t, u;
    /* Code that uses registers T5-T10, S11-S15; */
    foo(p, q, r, s, t);  /* Call foo */
    /* Code that reuses registers T9, T10; */

}
```

Here is the stack when bar is executing, clearly indicate in the spaces provided which procedure (main, foo, bar) saved specific entries on the stack.

| main | foo | bar | |
|---|---|---|---|
| _____ | _____ | _____ | p |
| _____ | _____ | _____ | q |
| _____ | _____ | _____ | r |
| _____ | _____ | _____ | s |
| _____ | _____ | _____ | t |
| _____ | _____ | _____ | u |
| _____ | _____ | _____ | T9 |
| _____ | _____ | _____ | T10 |
| _____ | _____ | _____ | p |
| _____ | _____ | _____ | q |
| _____ | _____ | _____ | r |
| _____ | _____ | _____ | s |
| _____ | _____ | _____ | t |
| _____ | _____ | _____ | x |

_____ _____ _____ y
_____ _____ _____ S11
_____ _____ _____ S12
_____ _____ _____ S13
_____ _____ _____ S14
_____ _____ _____ S15
_____ _____ _____ T6
_____ _____ _____ x
_____ _____ _____ y
_____ _____ _____ S11
_____ _____ _____ S12
_____ _____ _____ S13 <----------- Top of Stack