CHAPTER 4

Programming Models

4.1 Objectives

This chapter describes the programming models used on two types of processors: stack-based architectures and general-purpose register architectures. We begin with a discussion of the types of operations provided by most processors. A description of stack-based architectures and general-purpose register architectures follow. Each architecture's description includes a sample instruction set for that type of processor that will form the basis for examples and exercises throughout the rest of this book. The chapter concludes with a comparison of the two programming models and a discussion of how stacks are used to implement procedure calls, even on general-purpose register architectures.

After completing this chapter, you should

1. Be familiar with the different types of operations provided on most processors.
2. Understand stack-based architectures and be able to write short assembly-language programs in the instruction set described in this chapter.
3. Understand general-purpose register architectures and be able to write short assembly-language programs for these architectures.
4. Be able to compare general-purpose register architectures and stack-based architectures and describe situations in which each style would be more appropriate.
5. Understand procedure calls and how they are implemented.

4.2 Introduction

In the last chapter, we described programs as a set of machine instructions, without giving much detail about what instructions are or how they are implemented. In this chapter, we will cover two programming models for processors: stack-based

architectures and general-purpose register (GPR) architectures. A processor's *programming model* defines how instructions access their operands and how instructions are described in the processor's assembly language, but not the set of operations that are provided by the processor. As we will see, processors with different programming models can provide very similar sets of operations but may require very different approaches to programming.

Figure 4-1 gives a high-level view of how instructions are executed that we will use in this chapter. Later chapters will give more detailed explanations of instruction execution. First, the processor fetches (reads) the instruction from memory. The address of the next instruction to be executed is stored in a special register known as the *program counter* (PC), which is sometimes called the *instruction pointer* (IP), so the processor can easily determine where it should look for the next instruction in memory.

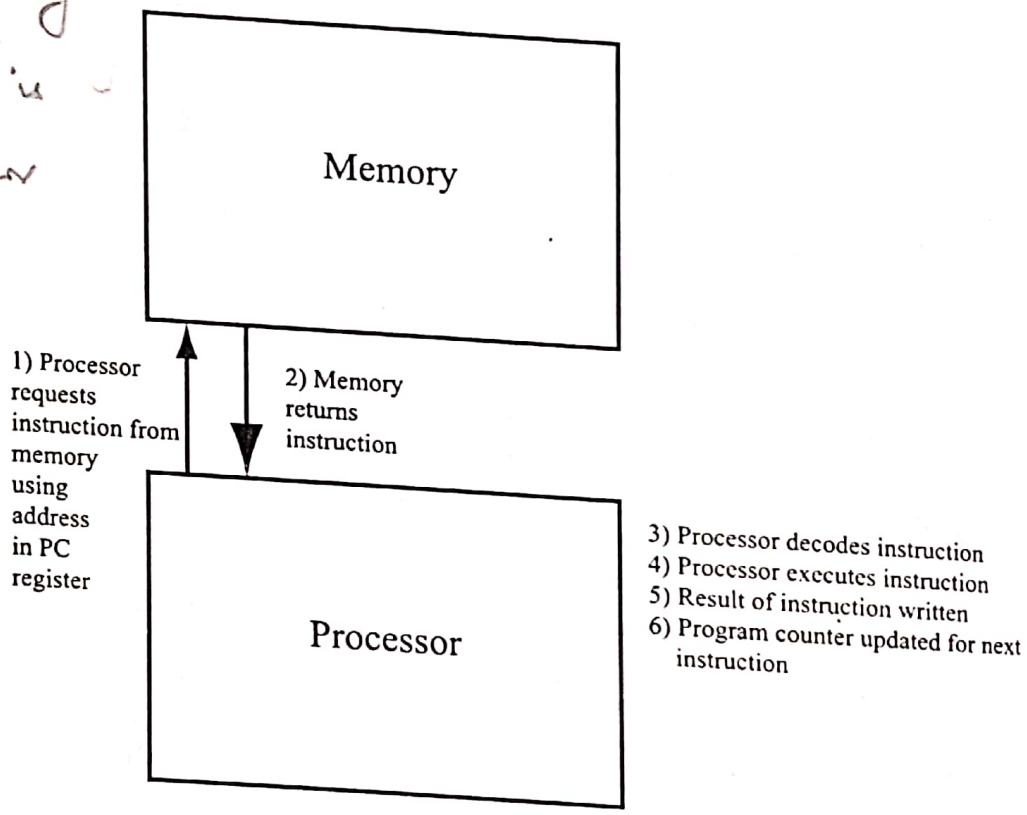


Fig. 4-1. Basic instruction execution.

Once the memory system has delivered the instruction to the processor, the processor examines the instruction to see what it has to do to complete the instruction, performs the operation specified by the instruction, and writes the result of the instruction into either a register or the memory system. The processor then updates the program counter to contain the address of the next instruction to be executed and repeats the process.

The remainder of this chapter begins with a discussion of the different types of instructions provided by most processors. We then give an introduction to stack-based architectures and present a sample instruction set for a stack-based archi-

tecture. A discussion of general-purpose register architectures follows, including a sample instruction set for these architectures. The chapter concludes with a discussion of how stacks are used to implement procedure calls on both stack-based and general-purpose register architectures.

4.3 Types of Instructions

One of the factors that differentiates processors is their ***instruction sets***—the sets of basic operations provided by each processor. Early computers had very different instruction sets, and instruction set design was one of the main tasks of computer architects. As the field has progressed, the set of operations provided by processors has converged, and now almost all processors provide very similar sets of instructions, regardless of whether they use a stack-based or general-purpose register programming model. These basic operations can be divided into four categories: arithmetic operations, memory operations, comparisons, and control operations (branches).

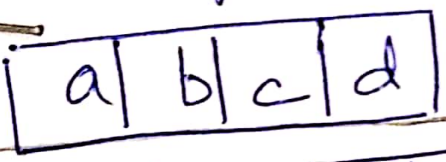
4.3.1 ARITHMETIC OPERATIONS

Arithmetic operations perform basic computations, such as additions, multiplications, logical operations (AND, OR), and data copying. They generally take one or two inputs, and generate one output. In general, arithmetic operations read their inputs from and write their outputs to the register file, though some CISC (complex instruction set computer) architectures allow arithmetic operations to reference memory. CISC architectures are covered in more detail in the next chapter.

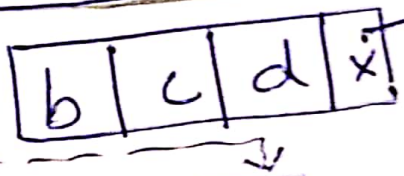
Figure 4-2 shows the set of arithmetic operations that we will be using in this book, which are representative of the arithmetic operations provided by most modern processors. Most processors provide a superset of these operations, often having multiple instructions that provide variations of a single operation. The operations presented here were selected as a compromise between completeness and complexity, with the goal of providing a rich enough instruction set to implement most programs without overwhelming the reader with complexity. Note that many of the instructions have integer and floating-point forms. This allows the hardware to determine whether it should treat the instruction's inputs as integers or floating-point quantities and determines which register file should be used on architectures that have separate integer and floating-point registers.

Most of the arithmetic operations presented in Fig. 4-2 are relatively self-explanatory, although two (ASH and LSH) deserve further explanation. These operations are examples of shift operations, operations that change the position of the bits in one of their inputs. Shift operations take the bits in their first input and move them left by a number of bit positions equal to the value of their second input (negative values of the second input indicate that the bits are shifted to the right). The differences between these operations lie in what values they insert into bit positions that are made vacant because a bit is shifted out of them but no bit is

A four bit register having four digits

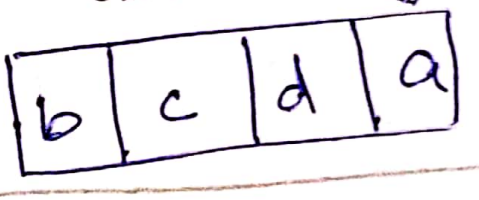


Shifting left :

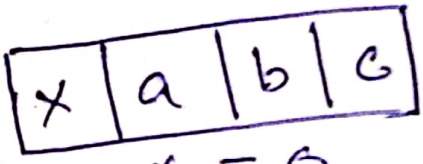


If shift is logical, put 0 in place of x.

Rotating



Shifting Right



for logical shift, $x = 0$

for Arithmetic shift, $x = a$

with rotate, $x = d$.

Logic shift is equivalent to

1. Multiplication by 2 when left shift
2. Division by 2 when right shift

This is true for integer xtions & ⁄ision.

Arithmetic shift is related to 2's Complement representation of Signed #s.

As Sign-bit is the left most bit, then the Arithmetic shift preserves the Sign

Shift is defined as either logical or arithmetic

Operation	Function
ADD	Adds its two integer operands
FADD	Adds its two floating-point operands
SUB	Subtracts its second integer operand from its first
FSUB	Subtracts its second floating-point operand from its first
MUL	Multiplies its two integer operands
FMUL	Multiplies its two floating-point operands
DIV	Divides its first integer input by its second
FDIV	Divides its first floating-point input by its second
MOV	Copies its input (of any type) to its output, both of which may be of any type
OR	Performs a logical OR on its two inputs, which can be of any type
AND	Performs a logical AND on its two inputs, which can be of any type
NOT	Performs logical negation on its input, which can be of any type
ASH	Shifts (arithmetic) its first input by the number of positions specified in its second input
LSH	Shifts (logical) its first input by the number of positions specified in its second input

Fig. 4-2. Arithmetic operations.

available to shift into them (for example, what value goes into the low bit of a word that is left-shifted one place).

Logical shift (LSH) operations are the simpler of the two shift operations. Bits that are shifted out of the word are discarded, and zeroes are shifted into vacant bit positions.

EXAMPLE

On a system with 8-bit data words, what is the result of doing an LSH operation whose first input is 25 and whose second input is 2?

Solution

The 8-bit integer representation of 25 is 0b 0001 1001. Shifting this to the left two bit positions gives 0b0110 01xx, where the "x" bits indicate vacant bits. The LSH operation specifies that zeroes are shifted into vacant bit positions, so the final result is 0b0110 0100, which is the 8-bit binary representation of 100.

One thing to note from the example is that shifting an unsigned or positive integer binary value to the left has the effect of multiplying it by 2^n, where n is the

Logic shift is defined as either left or right shift with zeroes inserted in emptied spaces
Arithmetic shift is defined as either left or right shift with only preserving sign bit

number of bit positions that the value is shifted by. Similarly, shifting an unsigned or positive integer binary value to the right n positions divides it by 2^n, discarding any bits of remainder from the division. Shift operations are often faster than multiplications and divisions, so many compilers and programmers use them in preference to multiplication operations when multiplying or dividing numbers by powers of 2.

However, the LSH operation does not generate the correct result for dividing a negative integer by a power of 2 when either two's-complement or sign-magnitude integers are used. For example, the 8-bit two's-complement representation of −16 is 0b1111 0000. Using LSH to shift this quantity right 1 bit (a shift of −1 positions) gives 0b0111 1000, the two's-complement representation of 120, because shifting a 0 into the high bit position of a two's-complement integer makes the result a positive value.

On systems that use two's-complement integers, the ASH (arithmetic shift) operation preserves the sign of the shifted value by copying the high bit of the word into all bits that are made vacant by a right-shift. When shifting to the left, zeroes are copied into the vacant bits to generate the correct result for multiplying by two. Thus, ASH can be used to multiply and divide two's-complement numbers by powers of 2, with the caveat that using ASH to divide negative numbers by powers of 2 rounds to the next most negative integer rather than discarding the remainder. For example, using ASH to divide −31 by 2 in a two's-complement system will yield a result of −16.

EXAMPLE

What is the result of executing an ASH operation whose first input is −15 and whose second input is 3, when executed on a system that uses 8-bit two's-complement integers? What if the second input is −3?

Solution

The 8-bit two's-complement representation of −15 is 0b1111 0001. Shifting to the left by three places, we get 0b1000 1000, because ASH places zeroes in vacant bit positions when shifting to the left. This is the 8-bit two's-complement representation of −120, the result of multiplying 15 by 2^3.

Shifting −15 by −3 positions is a right-shift of three places, yielding a result of 0b1111 1110, because ASH copies the high bit of the number being shifted into the vacant bit positions when doing a right-shift. This is the 8-bit two's-complement representation for −2, the correct result of dividing 15 by 2^3 if we round to the next most negative integer.

ASH can be implemented on systems that use sign-magnitude numbers by keeping the value of the sign bit the same as it was on the input word and performing a logical shift on the magnitude portion of the word. When ASH is implemented in this fashion, division operations on both positive and negative integers round their result by discarding the remainder from the operation.

4.3.2 MEMORY OPERATIONS

Memory operations transfer data between the processor and the memory system. As will be discussed in Chapter 5, one of the big differences between RISC (reduced instruction set computer) and CISC architectures is whether memory can only be accessed through memory operations or whether other operations can access the memory system. For this book, we will assume that processors have two memory operations: load (LD) and store (ST).

Each of these operations operates on an amount of data equal to the word size of the machine, so a LD reads one word of data from the addresses starting at the address specified by its input, and a ST writes one word of data to the memory starting at the address specified by its address input. Many processors provide a larger set of load and store operations that operate on different amounts of data, but we will stick with just two for simplicity. See Fig. 4-3.

Operation	Function
LD	Loads the contents of the address specified by its input operand into its destination
ST	Stores the value contained in its second input into the address specified by its first input.

Fig. 4-3. Memory operations.

4.3.3 COMPARISONS

As their name would suggest, comparison operations compare two or more values so that the program can make decisions. There is a great deal of variation among different processors in how they handle the output of comparison operations. Some processors write the results of the comparison into a register in the register file. Others provide a special register that holds the result of the most recent comparison operation. Architectures that use a special comparison result register have become less common in the recent past, as only having one place to put comparison results makes it impossible to perform multiple comparisons in parallel. Figure 4-4 shows a common set of comparison operations. Most processors also provide an equivalent set of floating-point comparison operations.

Operation	Function
EQ	Tests its two integer operands to see if they are equal
NEQ	Tests its two integer operands to see if they are not equal.
GT	Determines if its first integer operand is greater than its second
LT	Determines if its first integer operand is less than its second
GEQ	Determines if its first integer operand is greater than or equal to its second
LEQ	Determines if its first integer operand is less than or equal to its second

Fig. 4-4. Comparison operations.

Comparison operations are generally used in conjunction with control operations to create a *conditional branch* that executes either one section of a program or another depending on the result of the comparison. This usage is so common that many processors provide conditional branch operations that combine a comparison and a branch into one instruction. For simplicity, the instruction sets presented later in this chapter for stack-based and general-purpose register architectures will omit the comparison instructions and provide only conditional branch instructions, since conditional branches are the most common use of comparisons.

4.3.4 CONTROL OPERATIONS

Control (branch) operations affect program flow by changing the processor's PC. When an operation other than a control operation executes, the hardware increments the program counter by the size of the instruction so that it now points to the next instruction in the program. For example, on an architecture in which instructions are 32 bits long, 4 would be added to the PC after each instruction was executed because 32 bits is 4 bytes.

When a control operation executes, the program counter is set to the input[1] of the control instruction, causing execution to jump to a different point in the program. Control operations, which are often called branches, can be divided into two categories: *unconditional* and *conditional*. Unconditional branches, also called jumps, always set the PC equal to their input when they execute. Conditional branches set the PC equal to their input if some condition, such as the result of a comparison, is true. Depending on the architecture, the comparison can be performed as part of the branch operation, or can have been performed by a separate instruction earlier in the program. Figure 4-5 shows a common set of control operations that we will assume our processors provide.

When programmers write assembly-language programs or compilers generate them from high-level language programs, they generally do not specify the destination addresses of branches as the address of the destination instruction in memory. Instead, instructions are labeled with text values, and branch instructions refer to these values. For example, in the (infinite) loop shown in Fig. 4-6, the label "loop_start" is associated with the instruction that immediately follows it. The branch instruction at the end of the loop uses the label "loop_start" as its target, indicating that the program should branch back to the instruction following the label after executing the branch. One of the duties of the assembler is to compute the address corresponding to each label and insert that address into any branch instruction that references the label. As we will discuss in more detail in the next chapter, these addresses are often expressed as offsets from the branch to its destination rather than the address of the destination in memory.

Using labels instead of numeric addresses has two advantages. First, it is much easier for the programmer to understand. Looking at the code example in Fig. 4-6, it

[1] Many processors provide different *addressing modes* for branch instructions that cause the branch instruction to perform a computation and set the PC to the result of the computation instead of just setting the PC to the value of its input. Addressing modes are described in more detail in the next chapter.

Operation	Function
BR (or JMP)	Sets the PC equal to the value of its input operand, causing the instruction at that address to be executed next.
BEQ	Sets the PC equal to the value of its first operand if its other two inputs are equal.
BNE	Sets the PC equal to the value of its first operand if its other two inputs are not equal.
BLT	Sets the PC equal to the value of its first operand if its second operand is less than its third operand.
BGT	Sets the PC equal to the value of its first operand if its second operand is greater than its third operand.
BLE	Sets the PC equal to the value of its first operand if its second operand is less than or equal to its third operand.
BGE	Sets the PC equal to the value of its first operand if its second operand is greater than or equal to its third operand.

Fig. 4-5. Control operations.

is clear where the target of the branch instruction is, even if you don't know anything about the architecture of the processor. If the target were specified by a numeric address, you would have to know how much space each instruction takes up to be able to figure out the destination of each branch, and even that would take some work. The second advantage of using labels is that the address corresponding to a label can change if the instructions before the label change. If numeric addresses were used instead of labels, the destination of each branch would have to be changed every time the number of instructions before the branch changed. Instead, the programmer or compiler uses labels to specify the destination of each branch, and the assembler calculates the address of each label when the program is assembled.

```
loop_start:
        (instruction)
        (instruction)
        (instruction)
        BR loop_start;
```

Fig. 4-6. Label example.

4.4 Stack-Based Architectures

In a stack-based architecture, the register file is invisible to the program. Instead, instructions read their operands from, and write their results to, a *stack*, a last-in-first-out (LIFO) data structure.

4.4.1 THE STACK

As illustrated in Fig. 4-7, a stack is a last-in-first-out (LIFO) data structure. The name *stack* comes from the fact that the data structure acts like a stack of plates or

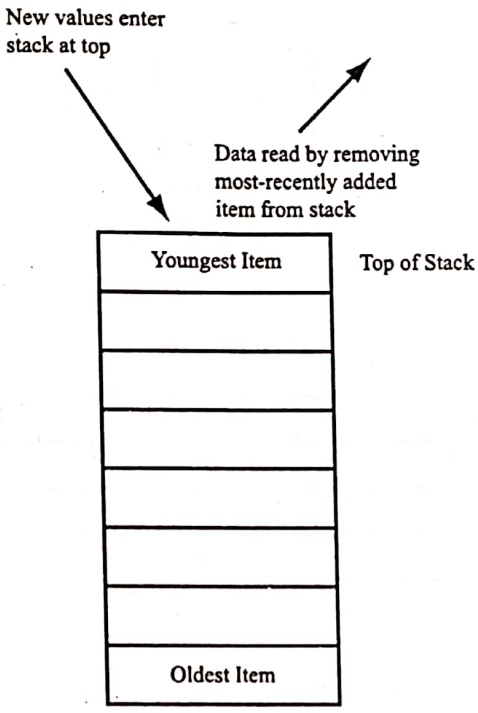


Fig. 4-7. A stack.

other items—when a new plate is put on a stack of plates, it goes on the top, and it is the first plate removed when someone takes a plate off of the stack. Stacks consist of a set of locations, each of which can hold one word of data. When a value is added to the stack, it is placed in the *top* location of the stack, and all data currently in the stack is moved down one location. Data can only be removed from the top of the stack. When this is done, all other data in the stack moves up one location. In general, data cannot be read from a stack without disturbing the stack, although some processors may provide special operations to allow this.

Stacks support two basic operations: PUSH and POP. A PUSH operation takes one argument and places the value of the argument on the top of the stack, pushing all previous data down one location. A POP operation removes the top value from the stack and returns it, allowing the value to be used as the input to an instruction. Figure 4-8 shows how a set of PUSH and POP operations affect a stack.

Initially, the stack is empty. The first PUSH operation places the value 4 in the top-of-stack location. The second PUSH operation places 5 in the top of the stack, pushing the 4 down to the next location. A POP operation is then executed, which removes the 5 from the top of the stack and returns it. The 4 then moves up to the top of the stack. Finally, a PUSH 7 operation is executed, leaving 7 in the top of the stack and 4 in the next location down.

4.4.2 IMPLEMENTING STACKS

As an abstract data structure, stacks are assumed to be infinitely deep, meaning that an arbitrary amount of data can be placed on the stack by the program. In practice,

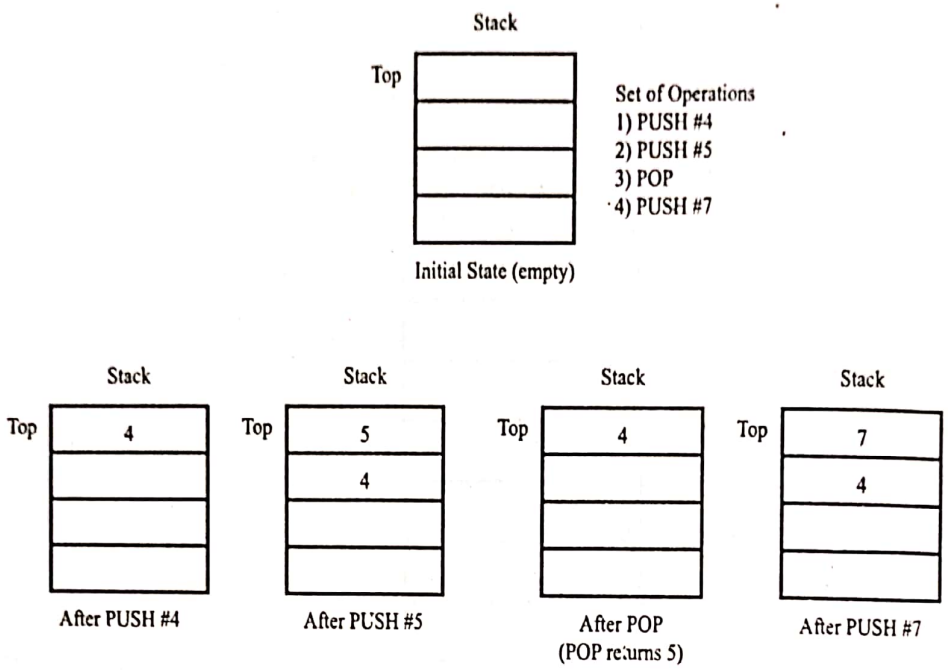


Fig. 4-8. Stack example.

stacks are implemented using buffers in memory, which are finite in size. If the amount of data in the stack exceeds the amount of space allocated to the stack, a *stack overflow* error occurs.

Figure 4-9 shows how a stack is implemented in the memory system of a computer. A fixed location defines the bottom of the stack, and a pointer gives the location of the top of the stack (the location of the last value pushed onto the stack). When a value is pushed onto the stack, the top-of-stack pointer is incremented by the word size of the machine and the value being pushed is stored into memory at the address pointed to by the new value of the top-of-stack pointer. To pop a value off of the stack, the value in the location pointed to by the top-of-stack pointer is read, and the top-of-stack pointer is decremented by the word size of the machine. When the top-of-stack pointer and the bottom pointer are the same, the stack is empty, and an attempt to pop data off of the stack results in an error. Several variations on this approach are possible, including ones where the bottom pointer points to the highest address in the stack buffer and the stack grows toward lower addresses.

This approach results in a completely functional stack, but accessing the stack tends to be relatively slow, because of the latency of the memory system. If the stack were kept completely in memory, executing a typical arithmetic instruction, such as ADD, would require four memory operations: one to fetch the instruction, two to fetch the operands from the stack, and one to write the result back onto the stack. To speed up stack accesses, stack-based processors may incorporate a register file in the processor and keep the top N values (where N is the size of the register file) on the stack in the register file.

Figure 4-10 illustrates how this works. Essentially, the register file is treated as a separate memory buffer from the buffer in main memory, with its own top-of-stack pointer. Since the register file contains a fixed number of storage locations, a

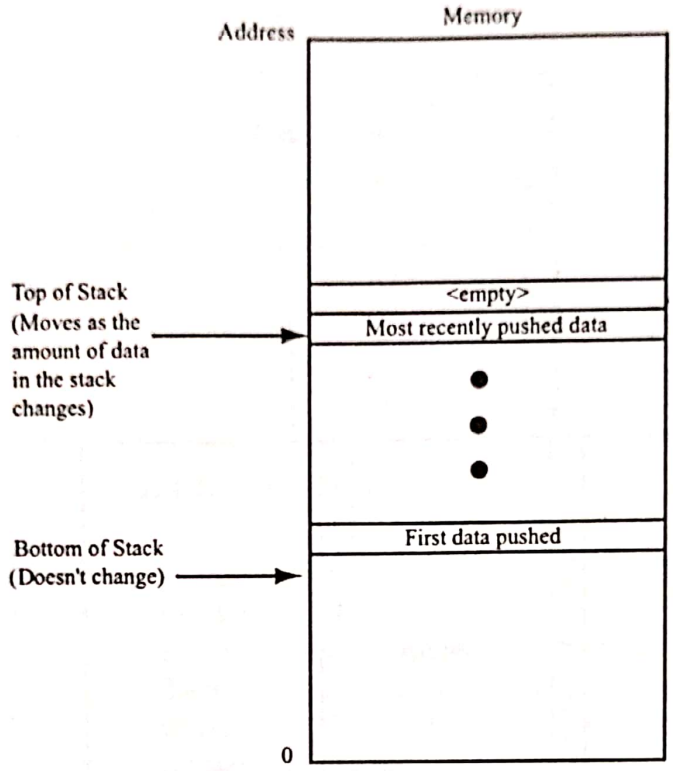


Fig. 4-9. Stack implementation in memory.

bottom-of-stack pointer is not required; the top-of-stack pointer simply keeps track of how many of the registers contain data. To push a value onto the stack, the register file top-of-stack pointer is incremented, and the value is copied into the register that it points to. When the register file becomes full, its contents are copied into the stack buffer in memory, and the top-of-stack pointers in both memory and the register file are adjusted to reflect that the register file is now empty and that the memory buffer contains more data. POP operations use a similar approach, with the register file being refilled from the buffer in memory when it becomes empty.

This approach can lead to a lot of memory accesses when the register file is nearly full or nearly empty and PUSHes and POPs alternate, because the register file is continually being copied to and from memory. Systems may choose to only half-empty or half-fill the register file when it over- or underflows to reduce this effect.

4.4.3 INSTRUCTIONS IN A STACK-BASED ARCHITECTURE

As stated above, instructions in a stack-based architecture get their operands from and write their results to the stack. When an instruction executes, it pops its operands off of the stack, performs the required computation, and pushes the result onto the top of the stack. Figure 4-11 shows an example of executing an ADD instruction on a stack-based architecture.

One of the significant advantages of stack-based architectures is that programs take up very little memory, because it is not necessary to specify where the source and destination of the operation are located. Thus, an instruction for a stack-based

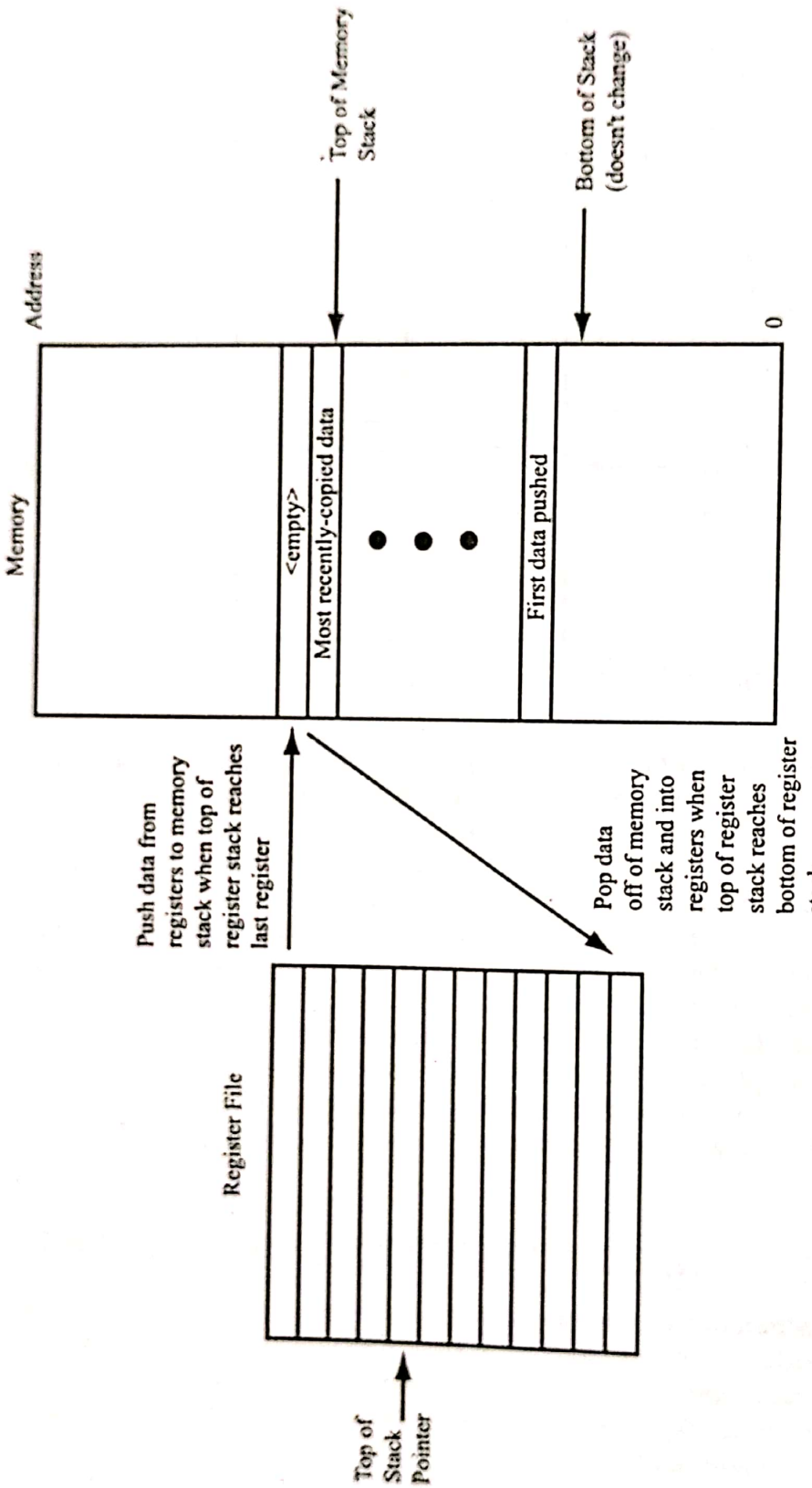


Fig. 4-10. Stack implementation using memory and registers.

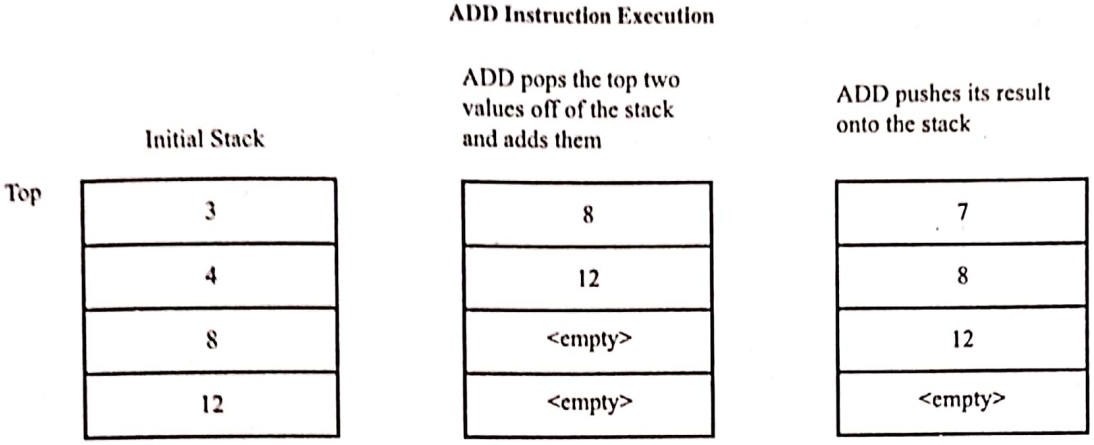


Fig. 4-11. Stack-based instruction execution.

processor can often be represented in just 1 byte that specifies the operation, although this can vary depending on how many operations the processor supports. The exception to this are PUSH instructions, whose operand is a constant specified in the instruction itself. Depending on the size of the constant allowed in the instruction, PUSH instructions can take 24 bits or more (8 bits to specify the operation and 16 bits of constant) to encode.

4.4.4 STACK-BASED INSTRUCTION SET

This section presents a sample instruction set for a stack-based processor, which we will use in programming exercises in this chapter. In describing this instruction set and the GPR instruction set presented later, we will use the following notation, which is frequently used in describing instructions:

"a <– b": The value of *b* is placed in *a*.
(a): The memory location whose address is contained in *a*.
#X: The constant value X.
PC: The program counter. We will only refer to the PC in describing branch instructions. For non-branch instructions, it will be assumed that the PC is incremented to point to the next instruction without specifying this in the instruction description. This is done to simplify the instruction descriptions, and because the amount by which the PC is incremented varies depending on how the instructions are encoded into binary values, which we are not specifying here.

Variables such as *a*, *b*, and *c* are used to assign temporary names to data popped off of the stack and do not necessarily correspond to actual storage locations in the processor. This notation will not be required in describing the GPR instruction set, but it is used here to make the order of operands clearer for operations where the order of operands affects the result, such as subtractions.

We will use the value STACK to indicate a reference to the stack. When STACK is used as an input, the stack is popped and the top value returned. When STACK is used as a destination, the value written is pushed onto the stack. Thus, the ADD operation can be described by the following sequence:

```
a <- STACK
b <- STACK
STACK <- a + b
```

For simplicity, we will assume that the processor implements conditional branch operations, rather than separate comparisons and branches. For operations where the order of operations matters, the order was chosen to match the order used on RPN calculators, so that the sequence PUSH #4, PUSH #3, SUB computes 4 – 3.

Our example processor will execute the following instructions:

PUSH #X	STACK <- X
POP	a <- STACK (the value popped is discarded)
LD	a <- STACK STACK <- (a)
ST	a <- STACK (a) <- STACK
ADD	a <- STACK b <- STACK STACK <- a + b (integer computation)
FADD	a <- STACK b <- STACK STACK <- a + b (floating-point computation)
SUB	a <- STACK b <- STACK STACK <- b - a (integer computation)
FSUB	a <- STACK b <- STACK STACK <- b - a (floating-point computation)
MUL	a <- STACK b <- STACK STACK <- a x b (integer computation)
FMUL	a <- STACK b <- STACK STACK <- a x b (floating-point computation)
DIV	a <- STACK b <- STACK STACK <- b / a (integer computation)
FDIV	a <- STACK b <- STACK STACK <- b / a (floating-point computation)
AND	a <- STACK b <- STACK STACK <- a & b (bit-wise computation)

```
OR      a <- STACK
        b <- STACK
        STACK <- a | b (bit-wise computation)
NOT     a <- STACK
        STACK <- !a (bit-wise negation)
ASH     a <- STACK
        b <- STACK
        STACK <- a shifted by b positions (arithmetic shift)
LSH     a <- STACK
        b <- STACK
        STACK <- a shifted by b positions (logical shift)
BR      PC <- STACK
BEQ     a <- STACK
        b <- STACK
        c <- STACK
        PC <- c if b is equal to a
BNE     a <- STACK
        b <- STACK
        c <- STACK
        PC <- c if b is not equal to a
BLT     a <- STACK
        b <- STACK
        c <- STACK
        PC <- c if b is less than a
BGT     a <- STACK
        b <- STACK
        c <- STACK
        PC <- c if b is greater than a
BLE     a <- STACK
        b <- STACK
        c <- STACK
        PC <- c if b is less than or equal to a
BGE     a <- STACK
        b <- STACK
        c <- STACK
        PC <- c if b is greater than or equal to a
```

4.4.5 PROGRAMS IN A STACK-BASED ARCHITECTURE

Stack-based programs are simply sequences of instructions that get executed one after the other. Given a stack-based program and an initial configuration of the stack, the result of the program can be computed by applying the first instruction in the program, determining the state of the stack and memory after the first instruction completes and repeating with each successive instruction.

Writing programs for stack-based processors can be somewhat more difficult, because stack-based processors are better suited to postfix (RPN) notation than traditional infix notation. Infix notation is the traditional way of representing mathematical expressions, in which the operation is placed between the operands.

In postfix notation, the operation is placed after the operands. For example, the infix expression "2 + 3" becomes "2 3 +" in postfix notation. Once an expression has been recoded in postfix notation, converting it into a stack-based program is straightforward. Starting at the left, each constant is replaced with a PUSH operation to place the constant on the stack, and operators are replaced with the appropriate instruction to perform their operation.

EXAMPLE
Create a stack-based program that performs the following computation:

$$2 + (7 \times 3)$$

Solution

First, we need to convert the expression into postfix notation. This is best done by iteratively converting each subexpression into postfix, so $2 + (7 \times 3)$ becomes $2 + (7\ 3\times)$ and then $2\ (7\ 3\times) +$. We then convert the postfix expression into a series of instructions, as described above, giving

```
PUSH #2
PUSH #7
PUSH #3
MUL
ADD
```

To verify that this program is correct, we hand-simulate its execution. After the three PUSH statements, the stack contains the values 3, 7, 2 (starting from the top of the stack). The MUL instruction pops the 3 and the 7 off the stack, multiplies them, and pushes the result (21) onto the stack, making the stack contents 21, 2. The ADD pops these two values off of the stack and adds them, placing the result (23) onto the stack, which is the result of the computation. This matches the result of the original expression, so the program is correct.

4.5 General-Purpose Register Architectures

In a general-purpose register (GPR) architecture, instructions read their operands from and write their results to a random-access register file, similar to the one illustrated in Fig. 4-12. The general-purpose register file allows an instruction to access the registers in any order by specifying the number (also called the register ID) of the register to be accessed, much like the memory system allows the addresses in the memory to be accessed in any order. Another significant difference between a general-purpose register file and a stack is that reading the contents of a general-purpose register does not change them, unlike popping values off of a stack. Successive reads of a general-purpose register with no intervening writes will return

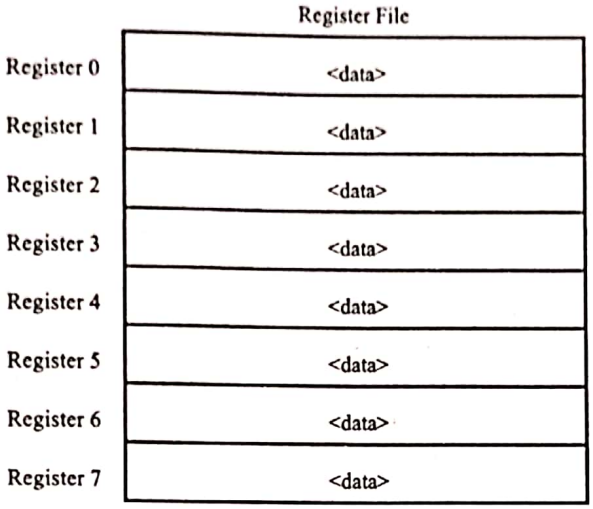


Fig. 4-12. General-purpose register file.

the same result, while successive stack pops will return the contents of the stack in LIFO order.

Many GPR architectures assign special meanings to some of the registers in the register file to make programming easier. For example, some processors hard-wire register 0 (r0) with the value 0 to make it easier to generate this common constant, and others make the program counter visible as one of the registers. The meanings are assigned by the hardware and cannot be changed by programs.

4.5.1 INSTRUCTIONS IN A GPR ARCHITECTURE

GPR processor instructions need to specify the registers that hold their input operands and the register where their result will be written. The most common format for this is the three-input instruction shown in Fig. 4-13. For most arithmetic instructions, the leftmost argument specifies the destination register of the instruction, while the other arguments specify the source registers[2]. Thus, the instruction ADD r1, r2, r3 instructs the processor to read the contents of registers r2 and r3, add them together, and write the result into r1. Formats for instructions that only have one input are also shown in the figure.

This instruction format is called "three-input" despite the fact that some operations have only two arguments to distinguish it from two-input instruction formats, in which one of the operand registers, such as the leftmost, is also the destination register. For example, the instruction ADD r1, r2 in a two-input instruction format tells the processor to add the contents of r1 and r2 and place the result in r1.

[2] Instruction encodings vary from processor to processor. In particular, some processors use the rightmost operand as their destination register, with other operands being the inputs to the instruction. We use the convention that the leftmost argument is the destination because that is the format used in many popular computer architecture texts.

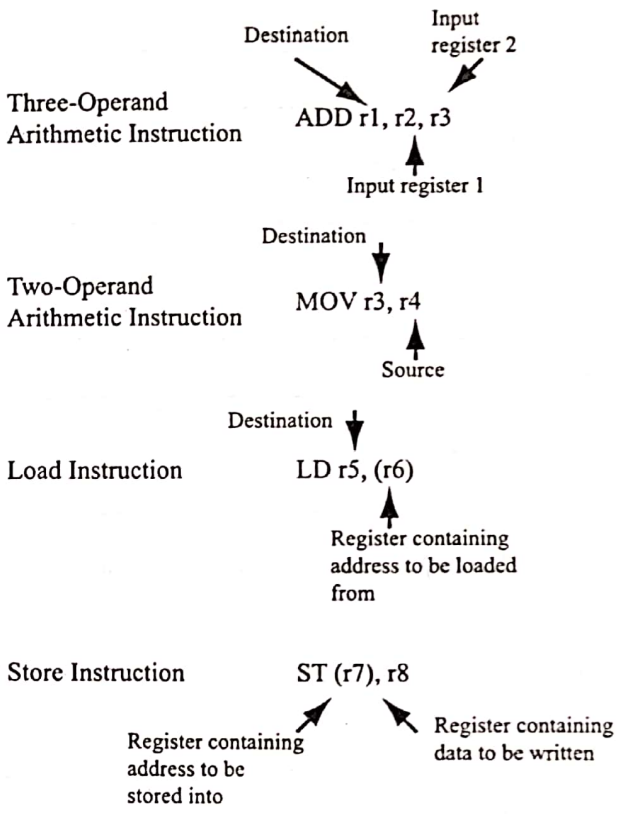


Fig. 4-13. Three-input instruction formats.

Three-input instruction formats are more flexible than two-input formats, as they allow an instruction's input and output registers to be selected independently but require more bits to encode. Many current architectures have 32 or more registers, requiring at least 5 bits to encode the register ID of each register referenced by the instruction. On 16-bit and smaller architectures, this makes it difficult to encode a three-input instruction in a single word of data, making two-input instructions attractive. On more modern 32- and 64-bit architectures, this is not as much of a problem, and virtually all of these architectures use three-input instruction encodings. Because they have become dominant, this book will assume three-input encodings for GPR instructions unless otherwise specified.

One significant difference between stack-based and GPR architectures is the fact that, in a GPR architecture, the program can choose which values should be stored in the register file at any given time, allowing the program to keep its most-accessed data in the register file. In contrast, the LIFO access restrictions on a stack limit the program's ability to choose what data is in the register file at any time. On early computers, it was felt that this was an advantage for stack architectures, as they would automatically tend to keep the most-referenced data in the register file that formed the top of the stack. As compiler technology has advanced, *register allocation* techniques that select the values that should be kept in the register file have improved to the point where GPR architectures can generally make better use of their register files than stack-based architectures, leading to GPR architectures having higher performance than stack-based architectures.

4.5.2 A GPR INSTRUCTION SET

This section presents a sample instruction set for a GPR processor that will be used for the rest of the examples in this book. This architecture will be described using the same notation used in Section 4.4.4, with the extension that the notation "rX" refers to register X when that register contains an integer value. The notation "fY" will be used to refer to register Y when that register contains a floating-point value. This notation is used because many processors use separate register files for integer and floating-point data.

Our GPR instruction set will implement the same operations as the stack-based instruction set presented earlier, except that the GPR instruction set does not contain the PUSH and POP operations, since these operations are only used to manipulate the stack. The GPR instruction set does, however, contain a MOV operation for copying data from one register to another. We will use a three-input instruction format in this book and will assume that one but not both of the source operands in any instruction may be a constant instead of a register, using the #X notation to denote constants. Our GPR instruction set contains the following operations:

LD ra, (rb)[3]	ra <- (rb) (ra may be a floating-point register)
ST (ra), rb	(ra) <- rb (rb may be a floating-point register)
MOV ra, rb	ra <- rb (ra and/or rb may be floating-point registers
ADD ra, rb, rc	ra <- rb + rc (integer computation)
FADD fa, fb, fc	fa <- fb + fc (floating-point computation)
SUB ra, rb, rc	ra <- rb - rc (integer computation)
FSUB fa, fb, fc	fa <- fb - fc (floating-point computation)
MUL ra, rb, rc	ra <- rb × rc (integer computation)
FMUL fa, fb, fc	fa <- fb × fc (floating-point computation)
DIV ra, rb, rc	ra <- rb / rc (integer computation)
FDIV fa, fb, fc	fa <- fb / fc (floating-point computation)
AND ra, rb, rc	ra <- rb & rc (bit-wise computation)
OR ra, rb, rc	ra <- rb | rc (bit-wise computation)
NOT ra, rb	ra <- !rb (bit-wise negation)
ASH ra, rb, rc	ra <- rb shifted by rc positions (arithmetic shift)
LSH ra, rb, rc	ra <- rb shifted by rc positions (logical shift)
BR ra	PC <- ra
BR label	PC <- label
BEQ ra, rb, rc	PC <- ra if rb is equal to rc

[3] Note that memory operations in our GPR architecture use parentheses around the register name that specifies the address they reference. This notation is used to be consistent with the addressing modes presented in the next chapter.

BEQ label, rb, rc	PC <- label if rb is equal to rc
BNE ra, rb, rc	PC <- ra if rb is not equal to rc
BNE label, rb, rc	PC <- label if rb is not equal to rc
BLT ra, rb, rc	PC <- ra if rb is less than rc
BLT label, rb, rc	PC <- label if rb is less than rc
BGT ra, rb, rc	PC <- ra if rb is greater than rc
BGT label, rb, rc	PC <- label if rb is greater than rc
BLE ra, rb, rc	PC <- ra if rb is less than or equal to rc
BLE label, rb, rc	PC <- label if rb is less than or equal to rc
BGE ra, rb, rc	PC <- ra if rb is greater than or equal to rc
BGE label, rb, rc	PC <- label if rb is greater than or equal to rc

4.5.3 PROGRAMS IN A GPR ARCHITECTURE

Like programs for stack-based architectures, programs for GPR architectures are simply a sequence of individual instructions. To figure out what a short instruction sequence does, apply each of the instructions in sequence, updating the contents of the register file after each instruction. Programming a GPR processor is less structured than programming a stack-based architecture, since there are fewer restrictions on the order in which operations must execute. On a stack-based processor, operations must execute in an order that leaves the operands for the next operation on the top of the stack. On a GPR processor, any order that places the operands for the next instruction in the register file before that instruction executes is valid—operations that reference different registers can be arbitrarily reordered without making the program incorrect. As we will see in later chapters, many modern processors take advantage of this to reorder instructions at runtime to improve performance.

EXAMPLE

Write a GPR program to compute the function $2 + (7 \times 3)$. Assume that the architecture has 16 registers, r0–r15, and that all registers contain 0 at the start of the program. The result of the computation may be left in any register.

Solution

A program that does this is

```
MOV r1, #7
MOV r2, #3
MUL r3, r1, r2
MOV r4, #2
ADD r4, r3, r4
```

The first two MOV instructions place the values 7 and 3 in r1 and r2, respectively. The MUL multiplies them together and places the result in r3. The next MOV places 2 in r4, and the final ADD adds this constant with the result of the MUL to generate the final result.

There are two things to note about this solution. First, the choice of registers used was arbitrary—any choice that didn't overwrite a register value before it was used would work. Second, the final instruction overwrites one of its operands. Since we don't need to use the value in r4 again, this is fine and was done to illustrate that it is possible.

4.6 Comparing Stack-Based and General-Purpose Register Architectures

Stack-based and general-purpose register architectures differ mainly in their interfaces to their register files. In stack-based architectures, data is stored on a stack in memory. The processor's file may be used to implement the top portion of the stack to allow faster access to the stack.

In GPR architectures, the register file is a random-access device where each register can be independently read and written by the processor. On these architectures, the register file and the memory are completely independent, and programs are responsible for moving data between these two types of storage as necessary.

Stack-based architectures were used in some early computer systems for two reasons. First, because the operands and destination of an instruction in a stack-based architecture are implicit, instructions take fewer bits to encode than they do in general-purpose register architectures. This reduced the amount of memory taken up by programs, which was a significant issue on early machines. Second, stack-based architectures manage registers automatically, freeing programmers from the need to decide which data should be kept in the register file.

Another advantage of stack-based architectures is that the instruction set does not change if the size of the register file changes. This means that programs written for a stack-based processor can be run on future versions of the processor that have more registers, and they will see performance improvements because they are able to keep more of the stack in registers. Stack-based architectures are also very easy to compile to—so easy that some compilers generate stack-based versions of a program as part of the compilation process even when compiling for a GPR processor.

General-purpose register architectures have become dominant in more recent years because of improvements in technology and the switch to high-level languages. As memory capacities have increased and memory costs have decreased, the amount of space taken up by a program has become less important, making stack-based architectures' advantage in instruction size less important. More significantly, compilers for GPR architectures are generally able to achieve better performance with a given number of general-purpose registers than they are on stack-based architectures with the same number of registers because the compiler

can choose which values to keep in the register file at any time and can use any value in the register file as the input to any instruction.

Because of their performance advantages and the decreasing importance of code size, virtually all recent workstation processors have been GPR architectures. Stack-based architectures are somewhat more attractive for embedded systems, in which low cost and low power consumption requirements often limit the amount of memory that can be included in a system, making code size more of a concern.

4.7 Using Stacks to Implement Procedure Calls

Procedure calls are an important part of virtually all computer languages. They allow commonly used functions to be written once and used whenever they are needed, and provide abstraction, making it easier for multiple people to collaborate on a program. However, several difficulties are involved in implementing procedure calls:

1. Programs need a way to pass inputs to procedures that they call and to receive outputs back from them.
2. Procedures need to be able to allocate space in memory for local variables without overwriting any data used by their calling program.
3. Since procedures may be called from many different locations within a program and are often compiled separately from the program that calls them, it is generally impossible to determine which registers may be safely used by a procedure and which contain data that will be needed after the procedure completes.
4. Procedures need a way to figure out where they were called from so that execution can return to the calling program when the procedure completes.

Most programming systems use a stack data structure to solve these problems. On GPR architectures, a stack is implemented in memory, such as the one illustrated in Fig. 4-9, while stack-based architectures can make use of the processor's main stack. When a procedure is called, a block of memory called a *stack frame* is allocated on the stack by incrementing the top-of-stack pointer by the number of locations in the stack frame. As illustrated in Fig. 4-14, a procedure's stack frame contains space for the contents of the calling program's register file, a pointer to the location that the procedure should branch to when it completes (its return address), the input arguments to the procedure, and the procedure's local variables.

When a procedure is called, the contents of the calling program's register file are copied into the stack frame, along with its return location and the inputs to the procedure. The procedure then uses the rest of the stack frame to hold its local variables. Since the number of input arguments and local variables varies from procedure to procedure, different procedures will have stack frames of different

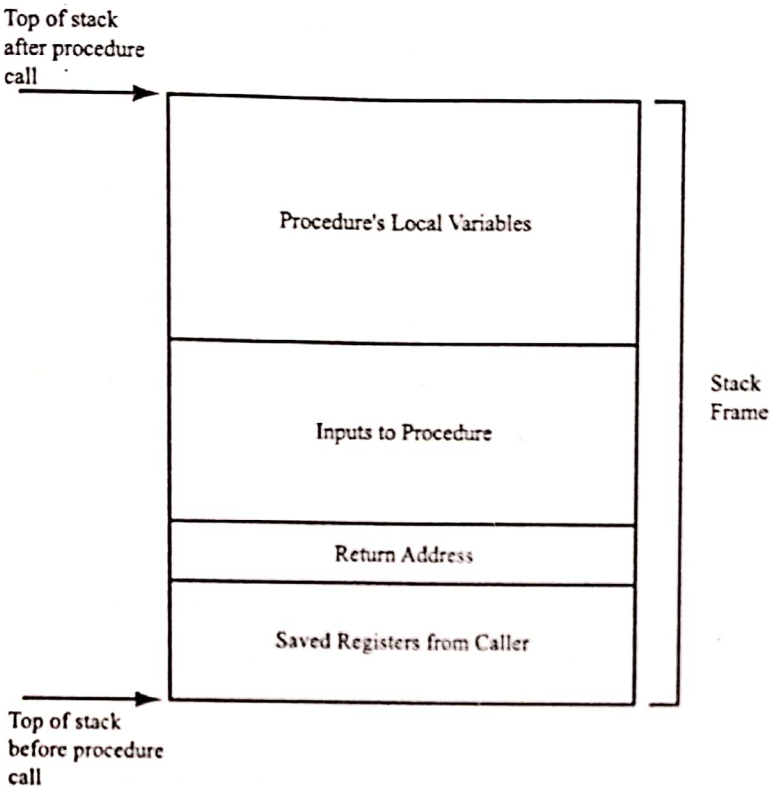


Fig. 4-14. Stack frame.

sizes. The arrangement of data within the stack frame also varies from programming system to programming system.

When a procedure finishes, it jumps to the return address contained in the stack frame, and execution of the calling program resumes. The calling program reads its saved register file contents out of the stack frame and handles the procedure's result, which can be passed either in a register or on the stack. Finally, the top-of-stack pointer is restored to its position before the procedure was called, popping the stack frame off of the stack.

When a program makes nested procedure calls (procedures that call other procedures), each nested procedure allocates its stack frame on top of those already on the stack. For example, Fig. 4-15 shows the contents of the stack during the execution of procedure h(), which was called from within procedure g(). Procedure g() was called from within f(), which was called by the main program. As long as the stack does not overflow, procedure calls can be nested as deeply as necessary and each stack frame will be popped off of the stack when execution returns to its calling program.

4.7.1 CALLING CONVENTIONS

Different programming systems may arrange the data in a procedure's stack frame differently and may require that the steps involved in calling a procedure be

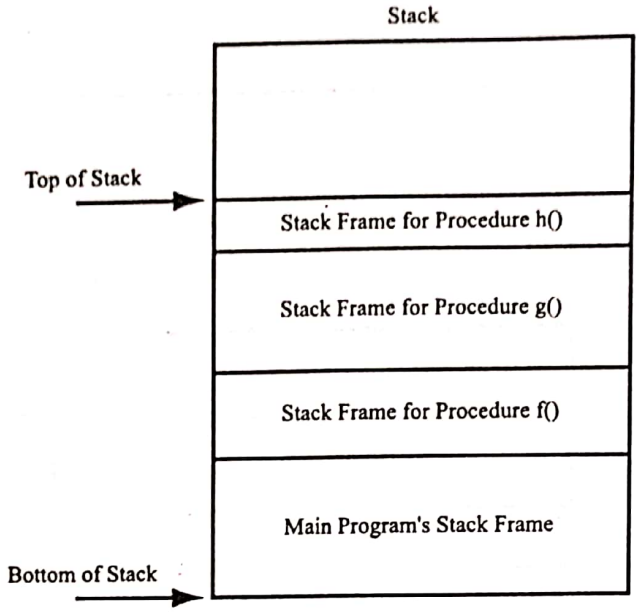


Fig. 4-15. Nested stack frames.

performed in different orders. The requirements that a programming system places on how a procedure is called and how data is passed between a calling program and its procedures are called *calling conventions*.

Many programming systems use their calling conventions to reduce the amount of data that needs to be copied to and from the stack during a procedure call. For example, a calling convention might specify a set of registers that are used to pass inputs and outputs between the calling program and the procedure. If a procedure's inputs and outputs fit in these registers, then it is not necessary to place them on the stack, reducing the number of memory references required. Programming systems generally also attempt to reduce the number of registers that must be saved and restored during a procedure call by identifying registers in the calling program whose values either will not be needed after the procedure call or whose values will not be overwritten by the procedure. These registers do not need to be saved or restored, reducing the cost of the procedure call.

4.8 Summary

This chapter has covered stack-based and general-purpose register architectures, two common programming models for processors. Stack-based processors use a last-in-first-out stack to hold the arguments to and results from their operations, while GPR architectures use a random-access register file. The two types of processors generally provide the same sorts of instructions, with a few exceptions, such as PUSH and POP instructions on a stack-based architecture.

Stack-based architectures were used in some early computer systems because they have very compact instruction encodings. Most instructions only need to

specify the operation to be performed, since the stack is the source of their operands and the destination of their results. A few operations, such as PUSH, take constant operands, which increases the number of bits required to encode the operation. Stack-based architectures are also very easy to compile for, and they allow compatibility between processors with different numbers of registers because the register file is part of the stack and is invisible to the program.

General-purpose register file architectures allow the program to select which values are kept in the random-access register file. Combined with the fact that, unlike popping a value off of a stack, reading a value from a GPR register file does not remove the value from the register file, this generally allows GPR programs to achieve higher performance than stack-based programs, because they require fewer instructions to complete a calculation. As the cost of memory has decreased, making the compactness of stack-based programs less significant, this improved performance has made general-purpose register architectures the dominant programming model.

The rest of this book will assume a GPR processor model, since this matches most current processors. In the next chapter, we will discuss some of the details of processor design, including the RISC versus CISC debate and register file design.

Solved Problems

Shift Operations

4.1. What is the result of the following operations when executed on an 8-bit processor that uses a two's-complement representation for negative integers?

a. LSH 14, 3
b. ASH 17, 5
c. LSH −23, −2
d. ASH −23, −2

Solution

a. The 8-bit two's-complement integer representation of 14 is 0b0000 1110. Shifting left by three positions gives 0b0111 0000, which is the integer representation for 112. $112 = 14 \times 2^3$, checking the correctness of the result.

b. The 8-bit two's-complement integer representation of 17 is 0b0001 0001. Shifting left by five positions gives 0b0010 0000, the integer representation of 32. Here, the result of the shift does not equal 17×2^5 because 544 cannot be represented as an 8-bit two's-complement integer.

c. The 8-bit two's-complement integer representation of −23 is 0b1110 1001. Shifting by −2 positions is a right-shift of two positions. Doing this with a logical shift operation gives a result of 0b0011 1010, the integer representation of 58, illustrating that LSH does not implement division by a power of 2 on negative integers.

d. Here, the only difference from the last part is that we're using an arithmetic shift operation, which copies the high bit of the shifted value into all vacant bits when a right-shift is done. This gives a result of 0b1111 1010, which is the representation of −6. Thus,

ASH does implement division by powers of 2 when right-shifting negative integers, although it rounds to the next most negative integer instead of dropping fractions.

Labels

4.2. Why is it generally more convenient to use labels than actual addresses to specify the destination of branch instructions?

Solution

Labels have two advantages. First, they allow the programmer to specify the destination of a branch in a way that does not change as the program changes. If the destination of a branch were specified as either the distance from the start of the program to the destination or as the distance from the branch instruction to the destination, then the programmer might have to change the destination of each branch in the program every time the number of instructions in the program changed. Second, they make it easier for humans to read assembly-language programs. A programmer can find the destination of a branch whose destination is specified using a label without knowing anything about how much memory each instruction takes up, simply by looking for the label in the program.

Stack-Based Architectures (I)

4.3. Briefly explain how instructions in a stack-based architecture access their operands.

Solution

Instructions in a stack-based architecture read their operands from and write their results to the stack. Operands are popped off of the stack in LIFO order, and the result of the instruction is pushed onto the top of the stack.

Stack-Based Architectures (II)

4.4. What is the maximum number of values on the stack at any time during the execution of the following sequence of PUSH and POP operations, and what are the contents of the stack after the sequence completes?

```
PUSH #1
PUSH #2
PUSH #3
POP
PUSH #4
POP
POP
```

Solution

The maximum number of values on the stack is 3, which occurs after the PUSH #3 and again after PUSH #4. At the end of the sequence, only the value 1 remains on the stack.

Stack Implementation

4.5. Why would it be a bad idea to implement a stack using just a processor's register file?

Solution

One of the benefits of a stack is that it presents the illusion of an infinitely large storage space to the programmer. Register files contain only a small number of storage locations, so a system that used only the register file to implement its stack would only be able to push a small amount of data onto the stack. Programmers using such a system would have to keep careful track of the amount of data on the stack at any time to avoid overflowing the stack, which would make the system much harder to program. Systems that allow the stack to expand into the memory when it overflows the register file are able to provide a much better approximation to the infinitely deep ideal stack.

Stack-Based Architecture Programming (I)

4.6. What value remains on the stack after the following sequence of instructions?

```
PUSH #4
PUSH #7
PUSH #8
ADD
PUSH #10
SUB
MUL
```

Solution

After the three PUSH operations, the stack contains 8, 7, 4 (starting at the top). The ADD pops the 8 and the 7, then pushes 15 on the stack, making the stack contents 15, 4. The PUSH 10 makes the stack 10, 15, 4. The SUB pops 10 and 15 from the stack, subtracts 10 from 15, and pushes 5 back on the stack. (Remember that SUB subtracts the top value on the stack from the next value down, so that PUSH x, PUSH y, SUB generates x–y.) Finally, the MUL pops 5 and 4 from the stack and pushes 20, leaving 20 on the stack.

Stack-Based Architecture Programming (II)

4.7. Write a stack-based program that computes the following function:
$5 + (3 \times 7) - 8$, assuming that the stack starts out empty.

Solution

First, we convert this expression to RPN, giving (5 (3 7×) +) 8 −). Note that the parentheses are completely unnecessary in RPN to generate the correct result. They are included solely to make parsing the RPN expression easier for the reader.

Then, the expression is translated into instructions:

```
PUSH #5
PUSH #3
PUSH #7
MUL
ADD
PUSH #8
SUB
```

Stack-Based Architecture Programming (III)

4.8. Assuming the stack starts out empty, write a stack-based program that computes $((10 \times 8) + (4 - 7))^2$

Solution

Because our processor does not provide an instruction to compute the square of a value, we need to compute $(10 \times 8) + (4 - 7)$ twice, so that the stack contains two copies of this result, and then multiply them. (It would also be possible to store the result into memory and load it into the stack twice.) As we will see in Problem 4.13, a GPR architecture can compute this much more efficiently by using the same register as both of the inputs to a MUL operation.

Transforming the computation into RPN and then into instructions gives the following program:

```
PUSH 10
PUSH 8
MUL
PUSH 4
PUSH 7
SUB
ADD [At this point, the stack contains only the first result of
(10 × 8) + (4 - 7)]
PUSH 10
PUSH 8
MUL
PUSH 4
PUSH 7
SUB
ADD [At this point, the stack contains two copies of
(10 × 8) + (4 - 7)]
MUL
```

GPR Architectures (I)

4.9. Briefly explain how instructions in a GPR architecture access their operands.

Solution

In a GPR architecture, instructions read their operands from and write their results to a random-access register file. Each instruction specifies both the registers containing its operands and the register where its result should be written.

GPR Architectures (II)

4.10. Briefly explain the difference between two-operand and three-operand instruction formats.

Solution

In a two-operand instruction format, one of the input registers to an instruction is also the instruction's output register. In a three-input instruction format, each instruction's input and output registers are specified independently. Three-operand instructions are more flexible than two-operand instructions, but they require more bits to encode.

GPR Programming (I)

4.11. Assuming that all registers start out containing 0, what is the value of r7 after the following instruction sequence is executed?

```
MOV r7, #4
MOV r8, #3
ADD r9, r7, r7
SUB r7, r9, r8
MUL r9, r7, r7
```

Solution

The two MOV instructions put the values 4 and 3 in r7 and r8, respectively. The ADD instruction adds 4 (the value of r7) to 4 (the value of r7), getting 8, and places that in r9. The SUB instruction subtracts 3 from 8, getting 5, and puts that in r7. Finally, the MUL instruction multiplies 5 and 5 to get 25, and puts that value in r9. Therefore, the value in r7 at the end of the instruction sequence is 5.

GPR Programming (II)

4.12. Write a GPR assembly-language program that performs the following computation, assuming that all registers start out containing 0. The final result may be left in any register.

$$5 + (3 \times 7) - 8.$$

Solution

Here is one such program, but there are many acceptable variations:

```
MOV r1, #5
MOV r2, #3
MOV r3, #7
MOV r4, #8
MUL r5, r2, r3
ADD r6, r1, r5
SUB r7, r6, r4
```

GPR Programming (III)

4.13. Assuming all registers start out containing 0, write a GPR program that computes $((10 \times 8) + (4 - 7))^2$. The final result may be left in any register.

Solution

Again, here is one of many programs that compute this function:

```
MOV r1, #10
MOV r2, #8
MOV r3, #4
MOV r4, #7
MUL r5, r1, r2
SUB r6, r3, r4
ADD r7, r5, r6
MUL r8, r7, r7
```

Comparing Stack-Based and GPR Architectures (I)

4.14. Give two advantages of stack-based architectures over GPR architectures.

Solution

This chapter has discussed three advantages of stack-based architectures:

1. Instructions in a stack-based architecture take up less memory than GPR architecture instructions, because stack-based architecture instructions do not have to specify the registers containing their sources or the register where their result should be written.
2. The register file in a stack-based architecture is invisible to the programmer, being the top part of the stack. As a result, future implementations of a stack-based architecture can contain different numbers of registers and still run programs written for the old processor. In contrast, the number of registers in a GPR architecture is encoded into the instruction set by the number of bits allocated for each register name, preventing programs written for a GPR processor from running on a GPR processor with a different number of registers.
3. The stack provides the illusion of an infinitely deep storage area, so programs do not have to worry about overflowing the amount of storage in the register file.

Listing any two of these advantages is a correct answer to the problem.

Comparing Stack-Based and GPR Architectures (II)

4.15. Give two advantages of GPR architectures over stack-based architectures.

Solution

The two main advantages of GPR architectures over stack-based architectures are as follows:

1. Reading a register in a GPR architecture does not affect its contents, while reading a value off of the top of the stack removes the value from the stack. When a given value is used more than once in a program, a GPR architecture can allocate that value to a register and read it repeatedly when it is needed. In contrast, stack-based architectures must either

use instructions to duplicate the value on the stack each time it is used as the input to an instruction or store the value in memory and reload it each time it is used.

2. GPR programs can choose which values to keep in the register file, while stack-based architectures are limited by the LIFO nature of the stack. Register allocation techniques in modern compilers are good enough at keeping the most-referenced values in the program in the register file that they require fewer memory references to complete a given program on a GPR architecture than on a stack-based architecture, improving performance.

Comparing Stack-Based and GPR Architectures (III)

4.16. Why have GPR architectures become dominant over stack-based architectures?

Solution

The key advantages of stack-based architectures are their smaller program size and lack of need for register allocation, while GPR architectures are capable of achieving better performance when the programmer/compiler does a good job of allocating values to registers. In early computers, memory was very expensive, so reducing program size was important. Also, many of the currently used register allocation techniques had not been developed.

As technology has advanced, memory has become cheap, making the reduced program size of stack-based architectures less important. Also, most programming is now done in high-level languages, and compilers contain sophisticated register allocation algorithms that make good use of the register file in a GPR architecture. Because of this, the performance advantages of GPR architectures have become more significant than the code size advantage of stack-based architectures, making GPR architectures better choices for most processor designs.

Stack Frames

4.17. A program is running on an architecture with 32 registers, each 32 bits wide. Addresses on this system are also 32 bits long. The program calls a procedure that takes four 32-bit arguments and allocates eight 32-bit internal variables. How large is the procedure's stack frame? (Assume that all of the calling program's registers must be saved.)

Solution

The stack frame has to be large enough to hold the caller's register file contents, the return address, the procedure's inputs, and its local variables. This is $(32 + 1 + 4 + 8) = 45$ 32-bit values for this procedure, or 180 bytes.