

MACHINE LANGUAGE AND ASSEMBLY LANGUAGE

In this chapter we tackle a central topic in computer architecture: the language understood by the computer's hardware, referred to as its **machine language**. The machine language is usually discussed in terms of its **assembly language**, which is functionally equivalent to the corresponding machine language except that the assembly language uses more intuitive names such as Move, Add, and Jump instead of the actual binary words of the language. (Programmers find constructs such as "Add r0, r1, r2" to be more easily understood and rendered without error than 0110101110101101.)

We begin by describing the **Instruction Set Architecture** (ISA) view of the machine and its operations. The ISA view corresponds to the Assembly Language/Machine Code level described in Figure 1-4: it is between the High Level Language view, where little or none of the machine hardware is visible or of concern, and the Control level, where machine instructions are interpreted as register transfer actions, at the Functional Unit level.

In order to describe the nature of assembly language and assembly language programming, we choose as a model architecture the **ARC** machine, which is a simplification of the commercial SPARC architecture common to Sun computers. (Additional architectural models are covered in *The Computer Architecture Companion* volume.)

We illustrate the utility of the various instruction classes with practical examples of assembly language programming, and we conclude with a Case Study of the Java bytecodes as an example of a common, portable assembly language that can be implemented using the native language of another machine.

4.1 Hardware Components of the Instruction Set Architecture

The ISA of a computer presents the assembly language programmer with a view of the machine that includes all the programmer-accessible hardware, and the instructions that manipulate data within the hardware. In this section we look at the hardware components as viewed by the assembly language programmer. We begin with a discussion of the system as a whole: the CPU interacting with its internal (main) memory and performing input and output with the outside world.

4.1.1 THE SYSTEM BUS MODEL REVISITED

Figure 4-1 revisits the system bus model that was introduced in Chapter 1.

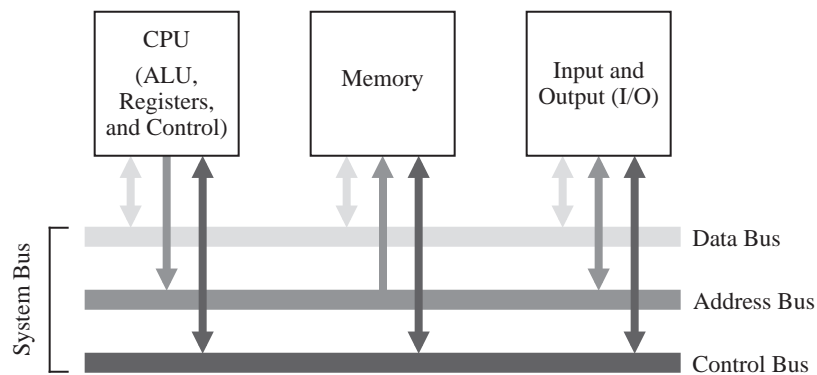


Figure 4-1 The system bus model of a computer system.

The purpose of the bus is to reduce the number of interconnections between the CPU and its subsystems. Rather than have separate communication paths between memory and each I/O device, the CPU is interconnected with its memory and I/O systems via a shared **system bus**. In more complex systems there may be separate busses between the CPU and memory and CPU and I/O.

Not all of the components are connected to the system bus in the same way. The CPU generates addresses that are placed onto the address bus, and the memory receives addresses from the address bus. The memory never generates addresses, and the CPU never receives addresses, and so there are no corresponding connections in those directions.

In a typical scenario, a user writes a high level program, which a compiler translates into assembly language. An assembler then translates the assembly language

program into machine code, which is stored on a disk. Prior to execution, the machine code program is loaded from the disk into the main memory by an operating system.

During program execution, each instruction is brought into the ALU from the memory, one instruction at a time, along with any data that is needed to execute the instruction. The output of the program is placed on a device such as a video display, or a disk. All of these operations are orchestrated by a control unit, which we will explore in detail in Chapter 6. Communication among the three components (CPU, Memory, and I/O) is handled with busses.

An important consideration is that the instructions are executed inside of the ALU, even though all of the instructions and data are initially stored in the memory. This means that instructions and data must be loaded from the memory into the ALU registers, and results must be stored back to the memory from the ALU registers.

4.1.2 MEMORY

Computer memory consists of a collection of consecutively numbered (addressed) registers, each one of which normally holds one byte. A **byte** is a collection of eight bits (sometimes referred to by those in the computer communications community as an **octet**). Each register has an address, referred to as a **memory location**. A **nibble**, or **nybble**, as it is sometimes spelled, refers to a collection of four adjacent bits. The meanings of the terms “bit,” “byte,” and “nibble” are generally agreed upon regardless of the specifics of an architecture, but the meaning of **word** depends upon the particular processor. Typical word sizes are 16, 32, 64, and 128 bits, with the 32-bit word size being the common form for ordinary computers these days, and the 64-bit word growing in popularity. In this text, words will be assumed to be 32-bits wide unless otherwise specified. A comparison of these data types is shown in Figure 4-2.

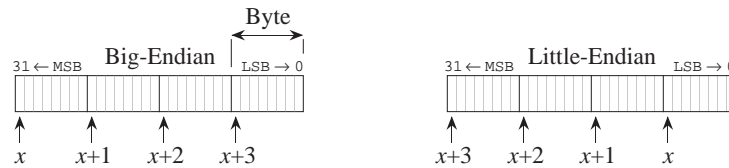
In a byte-addressable machine, the smallest object that can be referenced in memory is the byte, however, there are usually instructions that read and write multi-byte words. Multi-byte words are stored as a sequence of bytes, addressed by the byte of the word that has the lowest address. Most machines today have instructions that can access bytes, half-words, words, and double-words.

When multi-byte words are used, there are two choices about the order in which the bytes are stored in memory: most significant byte at lowest address, referred

Bit	0
Nibble	0110
Byte	10110000
16-bit word (halfword)	11001001 01000110
32-bit word	10110100 00110101 10011001 01011000
64-bit word (double)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101
128-bit word (quad)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101 00001011 10100110 11110010 11100110 10100100 01000100 10100101 01010001

Figure 4-2 Common sizes for data types.

to as **big-endian**, or least significant byte stored at lowest address, referred to as **little-endian**. The term “endian” comes from the issue of whether eggs should be broken on the big or little end, which caused a war by bickering politicians in Jonathan Swift’s *Gulliver’s Travels*. Examples of big and little-endian formats for a 4-byte, 32-bit word is illustrated in Figure 4-3.



Word address is x for both big-endian and little-endian formats.

Figure 4-3 Big-endian and little-endian formats.

Memory locations are arranged linearly in consecutive order as shown in Figure 4-3. Each of the numbered locations corresponds to a specific stored word (a word is composed of four bytes here). The unique number that identifies each word is referred to as its **address**. Since addresses are counted in sequence beginning with zero, the highest address is one less than the size of the memory. The highest address for a 2^{32} byte memory is $2^{32}-1$. The lowest address is 0.

The example memory that we will use for the remainder of the chapter is shown in Figure 4-4. This memory has a 32-bit **address space**, which means that a program can access a byte of memory anywhere in the range from 0 to $2^{32}-1$. The address space for our example architecture is divided into distinct regions which are used for the operating system, input and output (I/O), user programs, and the system stack, which comprise the **memory map**, as shown in Figure 4-3. The

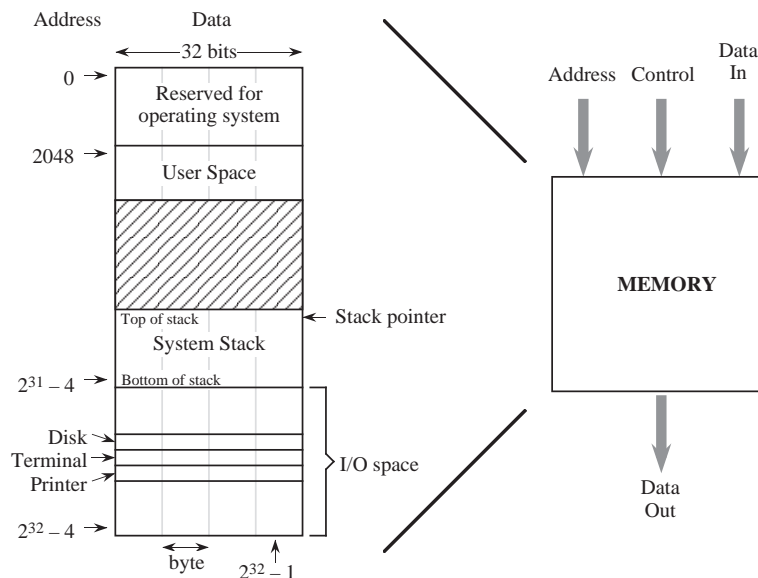


Figure 4-4 A memory map for an example architecture (not drawn to scale).

memory map differs from one implementation to another, which is partly why programs compiled for the same type of processor may not be compatible across systems.

The lower $2^{11} = 2048$ addresses of the memory map are reserved for use by the operating system. The user space is where a user's assembled program is loaded, and can grow during operation from location 2048 until it meets up with the system stack. The system stack starts at location $2^{31} - 4$ and grows toward lower addresses. The portion of the address space between 2^{31} and $2^{32} - 1$ is reserved for I/O devices. The memory map is thus not entirely composed of real memory, and in fact there may be large gaps where neither real memory nor I/O devices exist. Since I/O devices are treated like memory locations, ordinary memory read and write commands can be used for reading and writing devices. This is referred to as **memory mapped I/O**.

It is important to keep the distinction clear between what is an address and what is data. An address in this example memory is 32 bits wide, and a word is also 32 bits wide, but they are not the same thing. An address is a pointer to a memory location, which holds data.

In this chapter we assume that the computer's memory is organized in a single address space. The term **address space** refers to the numerical range of memory addresses to which the CPU can refer. In Chapter 7 (Memory), we will see that there are other ways that memory can be organized, but for now, we assume that memory as seen by the CPU has a single range of addresses. What decides the size of that range? It is the size of a memory address that the CPU can place on the address bus during read and write operations. A memory address that is n bits wide can specify one of 2^n items. This memory could be referred to as having an n -bit address space, or equivalently as having a (2^n) byte address space. For example, a machine having a 32-bit address space will have a maximum capacity of 2^{32} (4 GB) of memory. The memory addresses will range from 0 to $2^{32} - 1$, which is 0 to 4,294,967,295 decimal, or in the easier to manipulate hexadecimal format, from 00000000H to FFFFFFFFH. (The 'H' indicates a hexadecimal number in many assembly languages.)

4.1.3 THE CPU

Now that we are familiar with the basic components of the system bus and memory, we are ready to explore the internals of the CPU. At a minimum, the CPU consists of a **data section** that contains registers and an ALU, and a **control section**, which interprets instructions and effects register transfers, as illustrated in Figure 4-5. The data section is also referred to as the **datapath**.

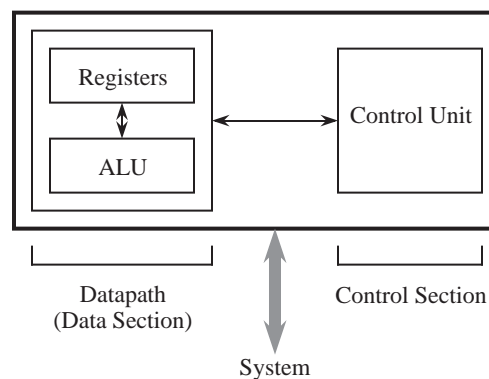


Figure 4-5 High level view of a CPU.

The control unit of a computer is responsible for executing the program instructions, which are stored in the main memory. (Here we will assume that the machine code is interpreted by the control unit one instruction at a time, though in Chapter 9 we shall see that many modern processors can process several

instructions simultaneously.) There are two registers that form the interface between the control unit and the data unit, known as the **program counter** (PC)[†] and the **instruction register** (IR). The PC contains the address of the instruction being executed. The instruction that is pointed to by the PC is fetched from the memory, and is stored in the IR where it is interpreted. The steps that the control unit carries out in executing a program are:

- 1) Fetch the next instruction to be executed from memory.
- 2) Decode the opcode.
- 3) Read operand(s) from main memory, if any.
- 4) Execute the instruction and store results.
- 5) Go to step 1.

This is known as the **fetch-execute cycle**.

The control unit is responsible for coordinating these different units in the execution of a computer program. It can be thought of as a form of a “computer within a computer” in the sense that it makes decisions as to how the rest of the machine behaves. We will treat the control unit in detail in Chapter 6.

The datapath is made up of a collection of registers known as the **register file** and the arithmetic and logic unit (ALU), as shown in Figure 4-6. The figure depicts the datapath of an example processor we will use in the remainder of the chapter.

The register file in the figure can be thought of as a small, fast memory, separate from the system memory, which is used for temporary storage during computation. Typical sizes for a register file range from a few to a few thousand registers. Like the system memory, each register in the register file is assigned an address in sequence starting from zero. These register “addresses” are much smaller than main memory addresses: a register file containing 32 registers would have only a 5-bit address, for example. The major differences between the register file and the system memory is that the register file is contained within the CPU, and is therefore much faster. An instruction that operates on data from the register file can often run ten times faster than the same instruction that operates on data in

[†] In Intel processors the program counter is called the instruction pointer, IP.

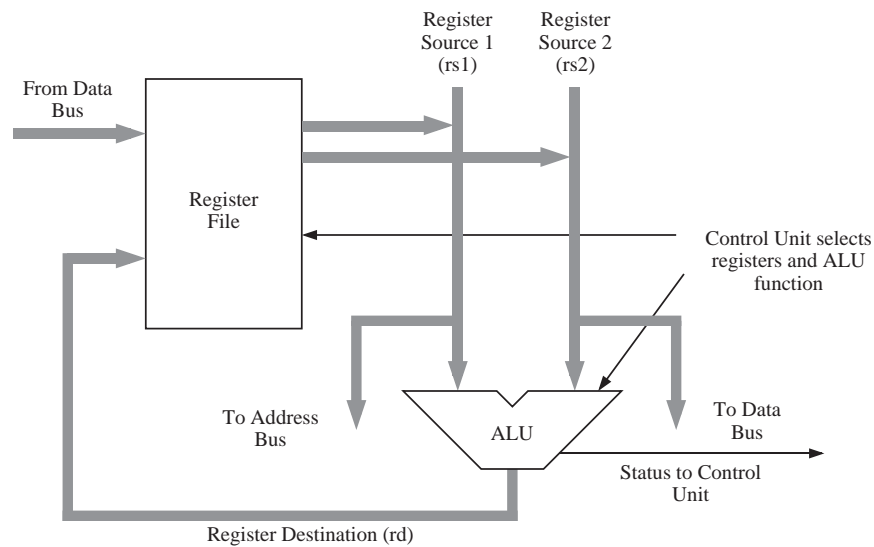


Figure 4-6 An example datapath.

memory. For this reason, register-intensive programs are faster than the equivalent memory intensive programs, even if it takes more register operations to do the same tasks that would require fewer operations with the operands located in memory.

Notice that there are several busses *inside* the datapath of Figure 4-6. Three busses connect the datapath to the system bus. This allows data to be transferred to and from main memory and the register file. Three additional busses connect the register file to the ALU. These busses allow two operands to be fetched from the register file simultaneously, which are operated on by the ALU, with the results returned to the register file.

The ALU implements a variety of binary (two-operand) and unary (one-operand) operations. Examples include add, and, not, or, and multiply. Operations and operands to be used during the operations are selected by the Control Unit. The two source operands are fetched from the register file onto busses labeled “Register Source 1 (rs1)” and “Register Source 2 (rs2).” The output from the ALU is placed on the bus labeled “Register Destination (rd),” where the results are conveyed back to the register file. In most systems these connections also include a path to the System Bus so that memory and devices can be accessed. This is shown as the three connections labeled “From Data Bus”, “To Data Bus”, and “To Address Bus.”

The Instruction Set

The **instruction set** is the collection of instructions that a processor can execute, and in effect, it defines the processor. The instruction sets for each processor type are completely different one from the other. They differ in the sizes of instructions, the kind of operations they allow, the type of operands they operate on, and the types of results they provide. This incompatibility in instruction sets is in stark contrast to the compatibility of higher level languages such as C, Pascal, and Ada. Programs written in these higher level languages can run almost unchanged on many different processors if they are **re-compiled** for the target processor.

(One exception to this incompatibility of machine languages is programs compiled into Java bytecodes, which are a machine language for a **virtual machine**. They will run unchanged on any processor that is running the Java Virtual Machine. The Java Virtual Machine, written in the assembly language of the target machine, intercepts each Java byte code and executes it as if it were running on a Java hardware (“real”) machine. See the Case Study at the end of the chapter for more details.)

Because of this incompatibility among instruction sets, computer systems are often identified by the type of CPU that is incorporated into the computer system. The instruction set determines the programs the system can execute and has a significant impact on performance. Programs compiled for an IBM PC (or compatible) system use the instruction set of an 80x86 CPU, where the ‘x’ is replaced with a digit that corresponds to the version, such as 80586, more commonly referred to as a Pentium processor. These programs will not run on an Apple Macintosh or an IBM RS6000 computer, since the Macintosh and IBM machines execute the instruction set of the Motorola **PowerPC** CPU. This does not mean that all computer systems that use the same CPU can execute the same programs, however. A PowerPC program written for the IBM RS6000 will not execute on the Macintosh without extensive modifications, however, because of differences in operating systems and I/O conventions.

We will cover one instruction set in detail later in the chapter.

Software for generating machine language programs

A **compiler** is a computer program that transforms programs written in a high-level language such as C, Pascal, or Fortran into machine language. Com-

compilers for the same high level language generally have the same “front end,” the part that recognizes statements in the high-level language. They will have different “back ends,” however, one for each target processor. The compiler’s back end is responsible for generating machine code for a specific target processor. On the other hand, the same program, compiled by different C compilers for the *same* machine can produce different compiled programs for the same source code, as we will see.

In the process of compiling a program (referred to as the **translation process**), a high-level source program is transformed into **assembly language**, and the assembly language is then translated into machine code for the target machine by an **assembler**. These translations take place at **compile time** and **assembly time**, respectively. The resulting object program can be linked with other object programs, at **link time**. The linked program, usually stored on a disk, is loaded into main memory, at **load time**, and executed by the CPU, at **run time**.

Although most code is written in high level languages, programmers may use assembly language for programs or fragments of programs that are time or space-critical. In addition, compilers may not be available for some special purpose processors, or their compilers may be inadequate to express the special operations which are required. In these cases also, the programmer may need to resort to programming in assembly language.

High level languages allow us to ignore the target computer architecture during coding. At the machine language level, however, the underlying architecture is the primary consideration. A program written in a high level language like C, Pascal, or Fortran may look the same and execute correctly after compilation on several different computer systems. The object code that the compiler produces for each machine, however, will be very different for each computer system, even if the systems use the same instruction set, such as programs compiled for the PowerPC but running on a Macintosh vs. running on an IBM RS6000.

Having discussed the system bus, main memory, and the CPU, we now examine details of a model instruction set, the ARC.

4.2 ARC, A RISC Computer

In the remainder of this chapter, we will study a model architecture that is based on the commercial Scalable Processor Architecture (**SPARC**) processor that was developed at Sun Microsystems in the mid-1980’s. The SPARC has become a

popular architecture since its introduction, which is partly due to its “open” nature: the full definition of the SPARC architecture is made readily available to the public (SPARC, 1992). In this chapter, we will look at just a subset of the SPARC, which we call “A RISC Computer” (**ARC**). “RISC” is yet another acronym, for **reduced instruction set computer**, which is discussed in Chapter 9. The ARC has most of the important features of the SPARC architecture, but without some of the more complex features that are present in a commercial processor.

4.2.1 ARC MEMORY

The ARC is a 32-bit machine with byte-addressable memory: it can manipulate 32-bit data types, but all data is stored in memory as bytes, and the address of a 32-bit word is the address of its byte that has the lowest address. As described earlier in the chapter in the context of Figure 4-4, the ARC has a 32-bit address space, in which our example architecture is divided into distinct regions for use by the operating system code, user program code, the system stack (used to store temporary data), and input and output, (I/O). These memory regions are detailed as follows:

- The lowest $2^{11} = 2048$ addresses of the memory map are reserved for use by the operating system.
- The user space is where a user’s assembled program is loaded, and can grow during operation from location 2048 until it meets up with the system stack.
- The system stack starts at location $2^{31} - 4$ and grows toward lower addresses. The reason for this organization of programs growing upward in memory and the system stack growing downward can be seen in Figure 4-4: it accommodates both large programs with small stacks and small programs with large stacks.
- The portion of the address space between 2^{31} and $2^{32} - 1$ is reserved for I/O devices—each device has a collection of memory addresses where its data is stored, which is referred to as “memory mapped I/O.”

The ARC has several data types (byte, halfword, integer, *etc.*), but for now we will consider only the 32-bit integer data type. Each integer is stored in memory as a collection of four bytes. ARC is a **big-endian** architecture, so the highest-order byte is stored at the lowest address. The largest possible byte address in the ARC is $2^{32} - 1$, so the address of the highest word in the memory map is

three bytes lower than this, or $2^{32} - 4$.

4.2.2 ARC INSTRUCTION SET

As we get into details of the ARC instruction set, let us start by making an overview of the CPU:

- The ARC has 32 32-bit general-purpose registers, as well as a PC and an IR.
- There is a **Processor Status Register** (PSR) that contains information about the state of the processor, including information about the results of arithmetic operations. The “arithmetic flags” in the PSR are called the **condition codes**. They specify whether a specified arithmetic operation resulted in a zero value (z), a negative value (n), a carry out from the 32-bit ALU (c), and an overflow (v). The v bit is set when the results of the arithmetic operation are too large to be handled by the ALU.
- All instructions are one word (32-bits) in size.
- The ARC is a **load-store** machine: the only allowable memory access operations load a value into one of the registers, or store a value contained in one of the registers into a memory location. All arithmetic operations operate on values that are contained in registers, and the results are placed in a register. There are approximately 200 instructions in the SPARC instruction set, upon which the ARC instruction set is based. A subset of 15 instructions is shown in Figure 4-7. Each instruction is represented by a **mnemonic**, which is a name that represents the instruction.

Data Movement Instructions

The first two instructions: **ld** (load) and **st** (store) transfer a word between the main memory and one of the ARC registers. These are the only instructions that can access memory in the ARC.

The **sethi** instruction sets the 22 most significant bits (MSBs) of a register with a 22-bit constant contained within the instruction. It is commonly used for constructing an arbitrary 32-bit constant in a register, in conjunction with another instruction that sets the low-order 10 bits of the register.

	Mnemonic	Meaning
Memory	ld	Load a register from memory
	st	Store a register into memory
Logic	sethi	Load the 22 most significant bits of a register
	andcc	Bitwise logical AND
	orcc	Bitwise logical OR
	orncc	Bitwise logical NOR
Arithmetic	srl	Shift right (logical)
	addcc	Add
	call	Call subroutine
Control	jmp1	Jump and link (return from subroutine call)
	be	Branch if equal
	bneg	Branch if negative
	bcs	Branch on carry
	bvs	Branch on overflow
	ba	Branch always

Figure 4-7 A subset of the instruction set for the ARC ISA.

Arithmetic and Logic Instructions

The **andcc**, **orcc**, and **orncc** instructions perform a bit-by-bit AND, OR, and NOR operation, respectively, on their operands. One of the two source operands must be in a register. The other may either be in a register, or it may be a 13-bit two's complement constant contained in the instruction, which is sign extended to 32-bits when it is used. The result is stored in a register.

For the **andcc** instruction, each bit of the result is set to 1 if the corresponding bits of both operands are 1, otherwise the result bit is set to 0. For the **orcc** instruction, each bit of the register is 1 if either or both of the corresponding source operand bits are 1, otherwise the corresponding result bit is set to 0. The **orncc** operation is the complement of **orcc**, so each bit of the result is 0 if either or both of the corresponding operand bits are 1, otherwise the result bit is set to 1. The “cc” suffixes specify that after performing the operation, the condition code bits in the PSR are updated to reflect the results of the operation. In particular, the *z* bit is set if the result register contains all zeros, the *n* bit is set if the most significant bit of the result register is a 1, and the *c* and *v* flags are cleared for these particular instructions. (Why?)

The shift instructions shift the contents of one register into another. The **srl** (shift right *logical*) instruction shifts a register to the right, and copies zeros into

the leftmost bit(s). The **sra** (shift right *arithmetic*) instruction (not shown), shifts the original register contents to the right, placing a copy of the MSB of the original register into the newly created vacant bit(s) in the left side of the register. This results in sign-extending the number, thus preserving its arithmetic sign.

The **addcc** instruction performs a 32-bit two's complement addition on its operands.

Control Instructions

The **call** and **jmp1** instructions form a pair that are used in calling and returning from a subroutine, respectively. **jmp1** is also used to transfer control to another part of the program.

The lower five instructions are called **conditional branch** instructions. The **be**, **bneg**, **bcs**, **bvs**, and **ba** instructions cause a branch in the execution of a program. They are called conditional because they test one or more of the condition code bits in the PSR, and branch if the bits indicate the condition is met. They are used in implementing high level constructs such as **goto**, **if-then-else** and **do-while**. Detailed descriptions of these instructions and examples of their usages are given in the sections that follow.

4.2.3 ARC ASSEMBLY LANGUAGE FORMAT

Each assembly language has its own syntax. We will follow the SPARC assembly language syntax, as shown in Figure 4-8. The format consists of four fields: an

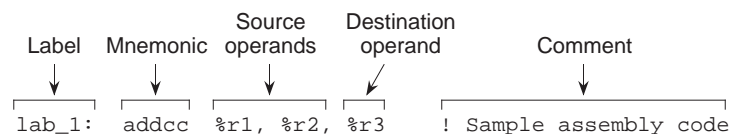


Figure 4-8 Format for a SPARC (as well as ARC) assembly language statement.

optional label field, an opcode field, one or more fields specifying the source and destination operands (if there are operands), and an optional comment field. A label consists of any combination of alphabetic or numeric characters, underscores (**_**), dollar signs (**\$**), or periods (**.**), as long as the first character is not a digit. A label must be followed by a colon. The language is sensitive to case, and so a distinction is made between upper and lower case letters. The language is “free format” in the sense that any field can begin in any column, but the relative

left-to-right ordering must be maintained.

The ARC architecture contains 32 registers labeled `%r0` – `%r31`, that each hold a 32-bit word. There is also a 32-bit Processor State Register (PSR) that describes the current state of the processor, and a 32-bit **program counter** (PC), that keeps track of the instruction being executed, as illustrated in Figure 4-9. The

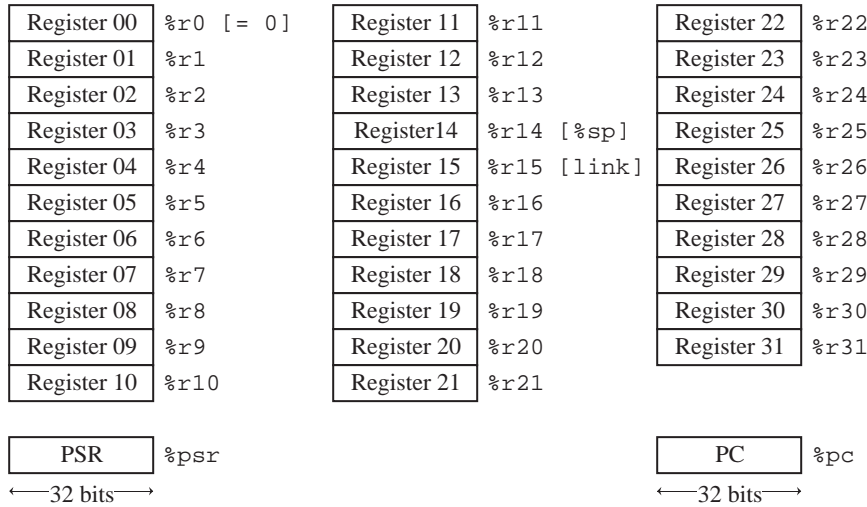


Figure 4-9 User-visible registers in the ARC.

PSR is labeled `%psr` and the PC register is labeled `%pc`. Register `%r0` always contains the value 0, which cannot be changed. Registers `%r14` and `%r15` have additional uses as a **stack pointer** (`%sp`) and a **link register**, respectively, as described later.

Operands in an assembly language statement are separated by commas, and the destination operand always appears in the rightmost position in the operand field. Thus, the example shown in Figure 4-8 specifies adding registers `%r1` and `%r2`, with the result placed in `%r3`. If `%r0` appears in the destination operand field instead of `%r3`, the result is discarded. The default base for a numeric operand is 10, so the assembly language statement:

```
addcc %r1, 12, %r3
```

shows an operand of $(12)_{10}$ that will be added to `%r1`, with the result placed in `%r3`. Numbers are interpreted in base 10 unless preceeded by “0x” or ending in “H”, either of which denotes a hexadecimal number. The comment field follows

the operand field, and begins with an exclamation mark ‘!’ and terminates at the end of the line.

4.2.4 ARC INSTRUCTION FORMATS

The **instruction format** defines how the various bit fields of an instruction are laid out by the assembler, and how they are interpreted by the ARC control unit. The ARC architecture has just a few instruction formats. The five formats are: **SETHI**, **Branch**, **Call**, **Arithmetic**, and **Memory**, as shown in Figure 4-10. Each

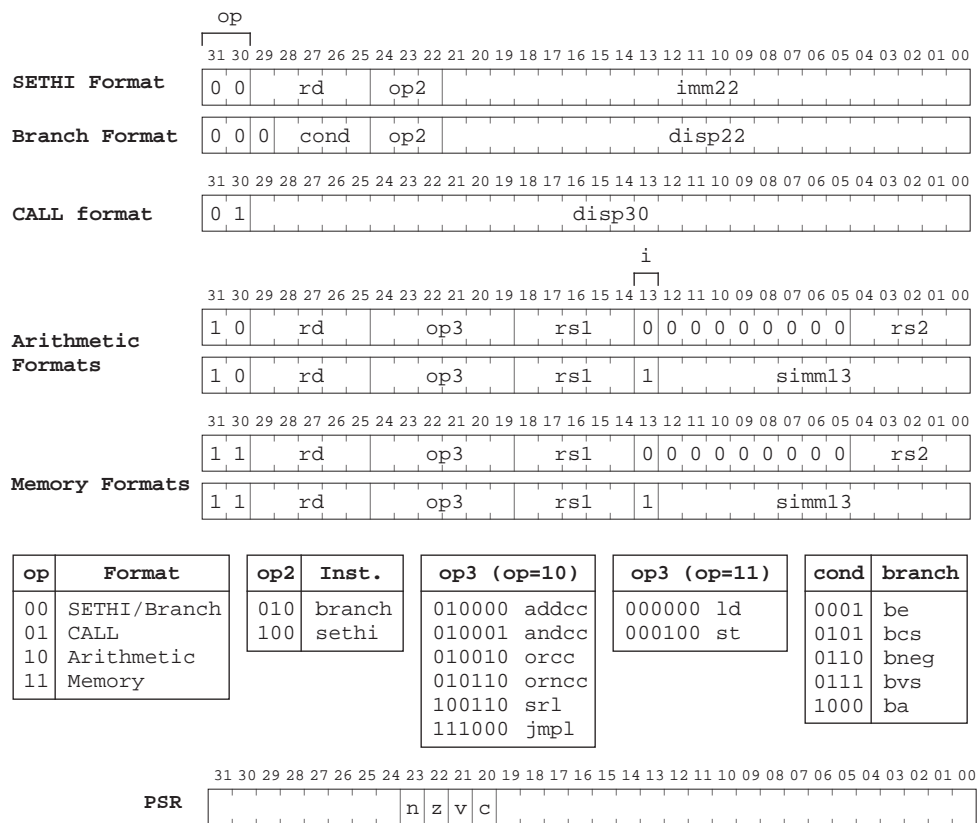


Figure 4-10 Instruction formats and PSR format for the ARC.

instruction has a mnemonic form such as “ld,” and an opcode. A particular instruction format may have more than one opcode field, which collectively identify an instruction in one of its various forms.

The leftmost two bits of each instruction form the `op` (opcode) field, which identifies the format. The SETHI and Branch formats both contain 00 in the `op` field, and so they can be considered together as the SETHI/Branch format. The actual SETHI or Branch format is determined by the bit pattern in the `op2` opcode field (010 = Branch; 100 = SETHI). Bit 29 in the Branch format always contains a zero. The five-bit `rd` field identifies the target register for the SETHI operation.

The `cond` field identifies the type of branch, based on the condition code bits (`n`, `z`, `v`, and `c`) in the PSR, as indicated at the bottom of Figure 4-10. The result of executing an instruction in which the mnemonic ends with “cc” sets the condition code bits such that `n`=1 if the result of the operation is negative; `z`=1 if the result is zero; `v`=1 if the operation causes an overflow; and `c`=1 if the operation produces a carry. The instructions that do not end in “cc” do not affect the condition codes. The `imm22` and `disp22` fields each hold a 22-bit constant that is used as the operand for the SETHI format (for `imm22`) or for calculating a displacement for a branch address (for `disp22`).

The CALL format contains only two fields: the `op` field, which contains the bit pattern 01, and the `disp30` field, which contains a 30-bit displacement that is used in calculating the address of the called routine.

The Arithmetic (`op` = 10) and Memory (`op` = 11) formats both make use of `rd` fields to identify either a source register for `st`, or a destination register for the remaining instructions. The `rs1` field identifies the first source register, and the `rs2` field identifies the second source register. The `op3` opcode field identifies the instruction according to the `op3` tables shown in Figure 4-10.

The `simm13` field is a 13-bit immediate value that is sign extended to 32 bits for the second source when the `i` (immediate) field is 1. The meaning of “sign extended” is that the leftmost bit of the `simm13` field (the sign bit) is copied to the left into the remaining bits that make up a 32-bit integer, before adding it to `rs1` in this case. This ensures that a two’s complement negative number remains negative (and a two’s complement positive number remains positive). For instance, $(-13)_{10} = (111111110011)_2$, and after sign extension to a 32-bit integer, we have $(111111111111111111111111110011)_2$ which is still equivalent to $(-13)_{10}$.

The Arithmetic instructions need two source operands and a destination operand, for a total of three operands. The Memory instructions only need two oper-

ands: one for the address and one for the data. The remaining source operand is also used for the address, however. The operands in the `rs1` and `rs2` fields are added to obtain the address when `i = 0`. When `i = 1`, then the `rs1` field and the `simml3` field are added to obtain the address. For the first few examples we will encounter, `%r0` will be used for `rs2` and so only the remaining source operand will be specified.

4.2.5 ARC DATA FORMATS

The ARC supports 12 different data formats as illustrated in Figure 4-11. The data formats are grouped into three types: signed integer, unsigned integer, and floating point. Within these types, allowable format widths are byte (8 bits), half-word (16 bits), word/singleword (32 bits), **tagged** word (32 bits, in which the two least significant bits form a **tag** and the most significant 30 bits form the value), **doubleword** (64 bits), and **quadword** (128 bits).

In reality, the ARC does not differentiate between unsigned and signed integers. Both are stored and manipulated as two's complement integers. It is their interpretation that varies. In particular one subset of the branch instructions assumes that the value(s) being compared are signed integers, while the other subset assumes they are unsigned. Likewise, the `c` bit indicates unsigned integer overflow, and the `v` bit, signed overflow.

The tagged word uses the two least significant bits to indicate **overflow**, in which an attempt is made to store a value that is larger than 30 bits into the allocated 30 bits of the 32-bit word. Tagged arithmetic operations are used in languages with dynamically typed data, such as Lisp and Smalltalk. In its generic form, a 1 in either bit of the tag field indicates an overflow situation for that word. The tags can be used to ensure proper alignment conditions (that words begin on four-byte boundaries, quadwords begin on eight-byte boundaries, *etc.*), particularly for pointers.

The floating point formats conform to the IEEE 754-1985 standard (see Chapter 2). There are special instructions that invoke the floating point formats that are not described here, that can be found in (SPARC, 1992).

4.2.6 ARC INSTRUCTION DESCRIPTIONS

Now that we know the instruction formats, we can create detailed descriptions of the 15 instructions listed in Figure 4-7, which are given below. The translation to

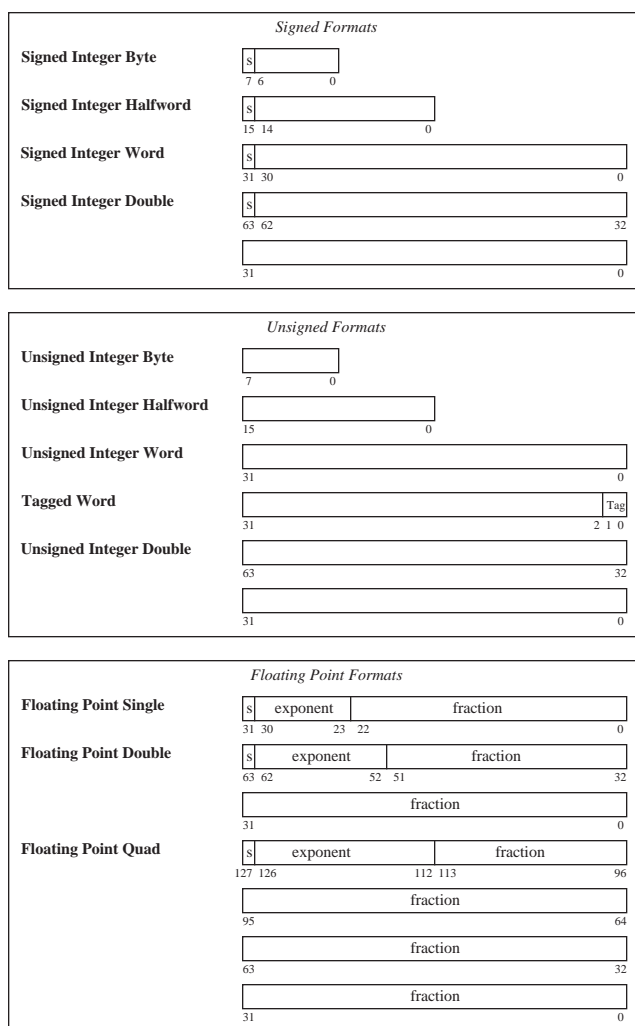


Figure 4-11 ARC data formats.

object code is provided only as a reference, and is described in detail in the next chapter. In the descriptions below, a reference to the *contents* of a memory location (for `ld` and `st`) is indicated by square brackets, as in “`ld [x], %r1`” which copies the contents of location `x` into `%r1`. A reference to the *address* of a memory location is specified directly, without brackets, as in “`call sub_r,`” which makes a call to subroutine `sub_r`. Only `ld` and `st` can access memory, therefore only `ld` and `st` use brackets. Registers are always referred to in terms of their contents, and never in terms of an address, and so there is no need to enclose references to registers in brackets.

Instruction: `ld`

Description: Load a register from main memory. The memory address must be aligned on a word boundary (that is, the address must be evenly divisible by 4). The address is computed by adding the contents of the register in the `rs1` field to either the contents of the register in the `rs2` field or the value in the `simm13` field, as appropriate for the context.

Example usage: `ld [x], %r1`
 or `ld [x], %r0, %r1`
 or `ld %r0+x, %r1`

Meaning: Copy the contents of memory location `x` into register `%r1`.

Object code: `11000010000000000010100000010000` (`x = 2064`)

Instruction: `st`

Description: Store a register into main memory. The memory address must be aligned on a word boundary. The address is computed by adding the contents of the register in the `rs1` field to either the contents of the register in the `rs2` field or the value in the `simm13` field, as appropriate for the context. The `rd` field of this instruction is actually used for the source register.

Example usage: `st %r1, [x]`

Meaning: Copy the contents of register `%r1` into memory location `x`.

Object code: `11000010001000000010100000010000` (`x = 2064`)

Instruction: `sethi`

Description: Set the high 22 bits and zero the low 10 bits of a register. If the operand is 0 and the register is `%r0`, then the instruction behaves as a **no-op (NOP)**, which means that no operation takes place.

Example usage: `sethi 0x304F15, %r1`

Meaning: Set the high 22 bits of `%r1` to $(304F15)_{16}$, and set the low 10 bits to zero.

Object code: `00000011001100000100111100010101`

Instruction: `andcc`

Description: Bitwise AND the source operands into the destination operand. The condition codes are set according to the result.

Example usage: `andcc %r1, %r2, %r3`

Meaning: Logically AND `%r1` and `%r2` and place the result in `%r3`.

Object code: `10000110100010000100000000000010`

Instruction: `orcc`

Description: Bitwise OR the source operands into the destination operand. The condition codes are set according to the result.

Example usage: `orcc %r1, 1, %r1`

Meaning: Set the least significant bit of `%r1` to 1.

Object code: `10000010100100000110000000000001`

Instruction: orncc

Description: Bitwise NOR the source operands into the destination operand. The condition codes are set according to the result.

Example usage: orncc %r1, %r0, %r1

Meaning: Complement %r1.

Object code: 10000010101100000100000000000000

Instruction: srl

Description: Shift a register to the right by 0 – 31 bits. The vacant bit positions in the left side of the shifted register are filled with 0's.

Example usage: srl %r1, 3, %r2

Meaning: Shift %r1 right by three bits and store in %r2. Zeros are copied into the three most significant bits of %r2.

Object code: 10000101001100000110000000000011

Instruction: addcc

Description: Add the source operands into the destination operand using two's complement arithmetic. The condition codes are set according to the result.

Example usage: addcc %r1, 5, %r1

Meaning: Add 5 to %r1.

Object code: 10000010100000000110000000000101

Instruction: call

Description: Call a subroutine and store the address of the current instruction (where the call itself is stored) in %r15, which effects a “call and link” operation. In the assembled code, the disp30 field in the CALL format will contain a 30-bit displacement from the address of the call instruction. The address of the next instruction to be executed is computed by adding $4 \times \text{disp30}$ (which shifts disp30 to the high 30 bits of the 32-bit address) to the address of the current instruction. Note that disp30 can be negative.

Example usage: call sub_r

Meaning: Call a subroutine that begins at location sub_r. For the object code shown below, sub_r is 25 words (100 bytes) farther in memory than the call instruction.

Object code: 0100000000000000000000000000011001

Instruction: jmp1

Description: Jump and link (return from subroutine). Jump to a new address and store the address of the current instruction (where the jmp1 instruction is located) in the destination register.

Example usage: jmp1 %r15 + 4, %r0

Meaning: Return from subroutine. The value of the PC for the call instruction was previously saved in %r15, and so the return address should be computed for the instruction that follows the call, at %r15 + 4. The current address is discarded in %r0.

Object code: 10000001110000111110000000000100

Instruction: `be`

Description: If the `z` condition code is 1, then branch to the address computed by adding $4 \times \text{disp22}$ in the Branch instruction format to the address of the current instruction. If the `z` condition code is 0, then control is transferred to the instruction that follows `be`.

Example usage: `be label`

Meaning: Branch to `label` if the `z` condition code is 1. For the object code shown below, `label` is five words (20 bytes) farther in memory than the `be` instruction.

Object code: 00000010100000000000000000000000101

Instruction: `bneg`

Description: If the `n` condition code is 1, then branch to the address computed by adding $4 \times \text{disp22}$ in the Branch instruction format to the address of the current instruction. If the `n` condition code is 0, then control is transferred to the instruction that follows `bneg`.

Example usage: `bneg label`

Meaning: Branch to `label` if the `n` condition code is 1. For the object code shown below, `label` is five words farther in memory than the `bneg` instruction.

Object code: 00001100100000000000000000000000101

Instruction: `bcs`

Description: If the `c` condition code is 1, then branch to the address computed by adding $4 \times \text{disp22}$ in the Branch instruction format to the address of the current instruction. If the `c` condition code is 0, then control is transferred to the instruction that follows `bcs`.

Example usage: `bcs label`

Meaning: Branch to `label` if the `c` condition code is 1. For the object code shown below, `label` is five words farther in memory than the `bcs` instruction.

Object code: 00001010100000000000000000000000101

Instruction: `bvs`

Description: If the `v` condition code is 1, then branch to the address computed by adding $4 \times \text{disp22}$ in the Branch instruction format to the address of the current instruction. If the `v` condition code is 0, then control is transferred to the instruction that follows `bvs`.

Example usage: `bvs label`

Meaning: Branch to `label` if the `v` condition code is 1. For the object code shown below, `label` is five words farther in memory than the `bvs` instruction.

Object code: 00001110100000000000000000000000101

Instruction: `ba`

Description: Branch to the address computed by adding $4 \times \text{disp22}$ in the Branch instruction format to the address of the current instruction.

Example usage: `ba label`

Meaning: Branch to `label` regardless of the settings of the condition codes. For the object code shown below, `label` is five words earlier in memory than the `ba` instruction.

Object code: 000100001011111111111111111111011

4.3 Pseudo-Ops

In addition to the ARC instructions that are supported by the architecture, there are also **pseudo-operations** (pseudo-ops) that are not opcodes at all, but rather instructions to the assembler to perform some action at assembly time. A list of pseudo-ops and examples of their usages are shown in Figure 4-12. Note that

Pseudo-Op	Usage	Meaning
<code>.equ</code>	<code>X .equ #10</code>	Treat symbol X as $(10)_{16}$
<code>.begin</code>	<code>.begin</code>	Start assembling
<code>.end</code>	<code>.end</code>	Stop assembling
<code>.org</code>	<code>.org 2048</code>	Change location counter to 2048
<code>.dwb</code>	<code>.dwb 25</code>	Reserve a block of 25 words
<code>.global</code>	<code>.global Y</code>	Y is used in another module
<code>.extern</code>	<code>.extern Z</code>	Z is defined in another module
<code>.macro</code>	<code>.macro M a, b, ...</code>	Define macro M with formal parameters a, b, ...
<code>.endmacro</code>	<code>.endmacro</code>	End of macro definition
<code>.if</code>	<code>.if <cond></code>	Assemble if <cond> is true
<code>.endif</code>	<code>.endif</code>	End of <code>.if</code> construct

Figure 4-12 Pseudo-ops for the ARC assembly language.

unlike processor opcodes, which are specific to a given machine, the kind and nature of the pseudo-ops are specific to a given *assembler*, because they are executed by the assembler itself.

The `.equ` pseudo-op instructs the assembler to *equate* a value or a character string with a symbol, so that the symbol can be used throughout a program as if the value or string is written in its place. The `.begin` and `.end` pseudo-ops tell the assembler when to start and stop assembling. Any statements that appear before `.begin` or after `.end` are ignored. A single program may have more than one `.begin/.end` pair, but there must be a `.end` for every `.begin`, and there must be at least one `.begin`. The use of `.begin` and `.end` are helpful in making portions of the program invisible to the assembler during debugging.

The `.org` (origin) pseudo-op causes the next instruction to be assembled with the assumption it will be placed in the specified memory location at runtime (location 2048 in Figure 4-12.) The `.dwb` (define word block) pseudo-op reserves a block of four-byte words, typically for an array. The **location counter** (which keeps track of which instruction is being assembled by the assembler) is moved ahead of the block according to the number of words specified by the argument to `.dwb` multiplied by 4.

The `.global` and `.extern` pseudo-ops deal with names of variables and addresses that are defined in one assembly code module and are used in another. The `.global` pseudo-op makes a label available for use in other modules. The `.extern` pseudo-op identifies a label that is used in the local module and is defined in another module (which should be marked with a `.global` in that module). We will see how `.global` and `.extern` are used when linking and loading are covered in the next chapter. The `.macro`, `.endmacro`, `.if`, and `.endif` pseudo-ops are also covered in the next chapter.

4.4 Examples of Assembly Language Programs

The process of writing an assembly language program is similar to the process of writing a high-level program, except that many of the details that are abstracted away in high-level programs are made explicit in assembly language programs. In this section, we take a look at two examples of ARC assembly language programs.

Program: Add Two Integers.

Consider writing an ARC assembly language program that adds the integers 15 and 9. One possible coding is shown in Figure 4-13. The program begins and

```
! This programs adds two numbers
    .begin
    .org 2048
prog1: ld    [x], %r1      ! Load x into %r1
      ld    [y], %r2      ! Load y into %r2
      addcc %r1, %r2, %r3  ! %r3 ← %r1 + %r2
      st    %r3, [z]      ! Store %r3 into z
      jmp1  %r15 + 4, %r0  ! Return
x:    15
y:    9
z:    0
      .end
```

Figure 4-13 An ARC assembly language program adds two integers.

ends with a `.begin/.end` pair. The `.org` pseudo-op instructs the assembler to begin assembling so that the assembled code is loaded into memory starting at location 2048. The operands 15 and 9 are stored in variables `x` and `y`, respectively. We can only add numbers that are stored in registers in the ARC (because only `ld` and `st` can access main memory), and so the program begins by loading registers `%r1` and `%r2` with `x` and `y`. The `addcc` instruction adds `%r1` and `%r2` and places the result in `%r3`. The `st` instruction then stores `%r3` in memory location `z`. The `jmp1` instruction with operands `%r15 + 4, %r0` causes a return to the next instruction in the calling routine, which is the operating system if this is the highest level of a user's program as we can assume it is here. The variables `x`, `y`, and `z` follow the program.

In practice, the SPARC code equivalent to the ARC code shown in Figure 4-13 is not entirely correct. The `ld`, `st`, and `jmp1` instructions all take at least two instruction cycles to complete, and since SPARC begins a new instruction at each clock tick, these instructions need to be followed by an instruction that does not rely on their results. This property of launching a new instruction before the previous one has completed is called **pipelining**, and is covered in more detail in Chapter 9.

Program: Sum an Array of Integers

Now consider a more complex program that sums an array of integers. One possible coding is shown in Figure 4-14. As in the previous example, the program begins and ends with a `.begin/.end` pair. The `.org` pseudo-op instructs the assembler to begin assembling so that the assembled code is loaded into memory starting at location 2048. A pseudo-operand is created for the symbol `a_start` which is assigned a value of 3000.

The program begins by loading the length of array `a`, which is given in bytes, into `%r1`. The program then loads the starting address of array `a` into `%r2`, and clears `%r3` which will hold the partial sum. Register `%r3` is cleared by ANDing it with `%r0`, which always holds the value 0. Register `%r0` can be ANDed with any register for that matter, and the result will still be zero.

The label `loop` begins a loop that adds successive elements of array `a` into the partial sum (`%r3`) on each iteration. The loop starts by checking if the number of remaining array elements to sum (`%r1`) is zero. It does this by ANDing `%r1` with itself, which has the side effect of setting the condition codes. We are interested in the `z` flag, which will be set to 1 if `%r1 = 0`. The remaining flags (`n`, `v`, and `c`)

```

! This program sums LENGTH numbers
! Register usage:      %r1 - Length of array a
!                      %r2 - Starting address of array a
!                      %r3 - The partial sum
!                      %r4 - Pointer into array a
!                      %r5 - Holds an element of a

        .begin                ! Start assembling
        .org 2048              ! Start program at 2048
a_start .equ 3000              ! Address of array a
        ld [length], %r1 ! %r1 ← length of array a
        ld [address], %r2 ! %r2 ← address of a
        andcc %r3, %r0, %r3 ! %r3 ← 0
loop:    andcc %r1, %r1, %r0 ! Test # remaining elements
        be done              ! Finished when length=0
        addcc %r1, -4, %r1 ! Decrement array length
        addcc %r1, %r2, %r4 ! Address of next element
        ld %r4, %r5 ! %r5 ← Memory[%r4]
        addcc %r3, %r5, %r3 ! Sum new element into r3
        ba loop              ! Repeat loop.

done:    jmpl %r15 + 4, %r0 ! Return to calling routine

length:    20                ! 5 numbers (20 bytes) in a
address:    a_start
        .org a_start        ! Start of array a
a:          25                ! length/4 values follow
           -10
           33
           -5
           7

        .end                ! Stop assembling

```

Figure 4-14 An ARC program sums five integers.

are set accordingly. The value of z is tested by making use of the `be` instruction. If there are no remaining array elements to sum, then the program branches to `done` which returns to the calling routine (which might be the operating system, if this is the top level of a user program).

If the loop is not exited after the test for $\%r1 = 0$, then $\%r1$ is decremented by the width of a word in bytes (4) by adding -4 . The starting address of array `a` (which is stored in $\%r2$) and the index into `a` ($\%r1$) are added into $\%r4$, which then points to a new element of `a`. The element pointed to by $\%r4$ is then loaded into $\%r5$, which is added into the partial sum ($\%r3$). The top of the loop is then revisited as a result of the “`ba loop`” statement. The variable `length` is stored after the instructions. The five elements of array `a` are placed in an area of memory according to the argument to the `.org` pseudo-op (location 3000).

Notice that there are three instructions for computing the address of the next

array element, given the address of the top element in `%r2`, and the length of the array in bytes in `%r1`:

```
addcc %r1, -4, %r1    ! Point to next element to be added
addcc %r1, %r2, %r4    ! Add it to the base of the array
ld %r4, %r5           ! Load the next element into %r5.
```

This technique of computing the address of a data value as the sum of a base plus an index is so frequently used that the ARC and most other assembly languages have special “addressing modes” to accomplish it. In the case of ARC, the `ld` instruction address is computed as the sum of two registers or a register plus a 13-bit constant. Recall that register `%r0` always contains the value zero, so by specifying `%r0` which is being done implicitly in the `ld` line above, we are wasting an opportunity to have the `ld` instruction itself perform the address calculation. A single register can hold the operand address, and we can accomplish in two instructions what takes three instructions in the example:

```
addcc %r1, -4, %r1    ! Point to next element to be added
ld %r1 + %r2, %r5     ! Load the next element into %r5.
```

Notice that we also save a register, `%r4`, which was used as a temporary place holder for the address.

4.4.1 VARIATIONS IN MACHINE ARCHITECTURES AND ADDRESSING

The ARC is typical of a load/store computer. Programs written for load/store machines generally execute faster, in part due to reducing CPU-memory traffic by loading operands into the CPU only once, and storing results only when the computation is complete. The increase in program memory size is usually considered to be a worthwhile price to pay.

Such was not the case when memories were orders of magnitude more expensive and CPUs were orders of magnitude smaller, as was the situation earlier in the computer age. Under those earlier conditions, for CPUs that had perhaps only a single register to hold arithmetic values, intermediate results had to be stored in memory. Machines had 3-address, 2-address, and 1-address arithmetic instructions. By this we mean that an instruction could do arithmetic with 3, 2, or 1 of its operands or results in memory, as opposed to the ARC, where all arithmetic and logic operands *must* be in registers.

Let us consider how the C expression $A = B * C + D$ might be evaluated by each of the three instruction types. In the examples below, when referring to a variable “A,” this actually means “the operand whose address is A.” In order to calculate some performance statistics for the program fragments below we will make the following assumptions:

- Addresses, opcodes, and data words are 16-bits – a not uncommon size in earlier machines.
- Opcodes are 8-bits in size.
- Operands are moved to and from memory one word at a time.

We will compute both program size, in bytes, and program memory traffic with these assumptions.

Memory traffic has two components: the code itself, which must be fetched from memory to the CPU in order to be executed, and the data values—operands must be moved into the CPU in order to be operated upon, and results moved back to memory when the computation is complete. Observing these computations allows us to visualize some of the trade-offs between program size and memory traffic that the various instruction classes offer.

Three-Address Instructions

In a 3-address instruction, the expression $A = B * C + D$ might be coded as:

```
mult  B, C, A
add   D, A, A
```

which means multiply B by C and store the result at A. (The `mult` and `add` operations are generic; they are not ARC instructions.) Then, add D to A (at this point in the program, A holds the temporary result of multiplying B times C) and store the result at address A. The program size is 8×2 or 16 bytes. Memory traffic is $16 + 2 \times 2 \times 3$ or 28 bytes.

Two Address Instructions

In a two-address instruction, one of the operands is overwritten by the result.

Here, the code for the expression $A = B * C + D$ is:

```
load  B, A
mult  C, A
add   D, A
```

The program size is now $3 \times 3 \times 2$ or 18 bytes. Memory traffic is $18 + 2 \times 2 + 2 \times 2 \times 3$ or 34 bytes.

One Address, or Accumulator Instructions

A one-address instruction employs a single arithmetic register in the CPU, known as the **accumulator**. The accumulator typically holds one arithmetic operand, and also serves as the target for the result of an arithmetic operation. The one-address format is not in common use these days, but was more common in the early days of computing when registers were more expensive and frequently served multiple purposes. It serves as temporary storage for one of the operands and also for the result. The code for the expression $A = B * C + D$ is now:

```
load  B
mult  C
add   D
store A
```

The `load` instruction loads B into the accumulator, `mult` multiplies C by the accumulator and stores the result in the accumulator, and `add` does the corresponding addition. The `store` instruction stores the accumulator in A. The program size is now $2 \times 2 \times 4$ or 16 bytes, and memory traffic is $16 + 4 \times 2$ or 24 bytes.

Special-Purpose Registers

In addition to the general-purpose registers and the accumulator described above, most modern architectures include other registers that are dedicated to specific purposes. Examples include

- Memory index registers: The Intel 80x86 Source Index (SI) and Destination Index (DI) registers. These are used to point to the beginning or end of an array in memory. Special “string” instructions transfer a byte or a

word from the starting memory location pointed to by SI to the ending memory location pointed to by DI, and then increment or decrement these registers to point to the next byte or word.

- **Floating point registers:** Many current-generation processors have special registers and instructions that handle floating point numbers.
- **Registers to support time, and timing operations:** The PowerPC 601 processor has Real-Time Clock registers that provide a high-resolution measure of real time for indicating the date and the time of day. They provide a range of approximately 135 years, with a resolution of 128 ns.
- **Registers in support of the operating system:** most modern processors have registers to support the memory system.
- **Registers that can be accessed only by “privileged instructions,” or when in “Supervisor mode.”** In order to prevent accidental or malicious damage to the system, many processors have special instructions and registers that are unavailable to the ordinary user and application program. These instructions and registers are used only by the operating system.

4.4.2 PERFORMANCE OF INSTRUCTION SET ARCHITECTURES

While the program size and memory usage statistics calculated above are observed out of context from the larger programs in which they would be contained, they do show that having even one temporary storage register in the CPU can have a significant effect on program performance. In fact, the Intel Pentium processor, considered among the faster of the general-purpose CPUs, has only a single accumulator, though it has a number of special-purpose registers that support it. There are many other factors that affect real-world performance of an instruction set, such as the time an instruction takes to perform its function, and the speed at which the processor can run.

4.5 Accessing Data in Memory—Addressing Modes

Up to this point, we have seen four ways of computing the address of a value in memory: (1) a constant value, known at assembly time, (2) the contents of a register, (3) the sum of two registers, and (4) the sum of a register and a constant.

Addressing Mode	Syntax	Meaning
-----------------	--------	---------

Table 4.1 Addressing Modes

Immediate	#K	K
Direct	K	M[K]
Indirect	(K)	M[M[K]]
Register	(Rn)	M[Rn]
Register Indexed	(Rm + Rn)	M[Rm + Rn]
Register Based	(Rm + X)	M[Rm + X]
Register Based Indexed	(Rm + Rn + X)	M[Rm + Rn + X]

Table 4.1 Addressing Modes

Table 4.1 gives names to these addressing modes, and shows a few others as well. Notice that the syntax of the table differs from that of the ARC. This is a common, unfortunate feature of assembly languages: each one differs from the rest in its syntax conventions. The notation $M[x]$ in the Meaning column assumes memory is an array, M , whose byte index is given by the address computation in brackets. There may seem to be a bewildering assortment of addressing modes, but each has its usage:

- Immediate addressing allows a reference to a constant that is known at assembly time.
- Direct addressing is used to access data items whose address is known at assembly time.
- Indirect addressing is used to access a pointer variable whose address is known at compile time. This addressing mode is seldom supported in modern processors because it requires two memory references to access the operand, making it a complicated instruction. Programmers who wish to access data in this form must use two instructions, one to access the pointer and another to access the value to which it refers. This has the beneficial side effect of exposing the complexity of the addressing mode, perhaps discouraging its use.
- Register indirect addressing is used when the address of the operand is not known until run time. Stack operands fit this description, and are accessed by register indirect addressing, often in the form of push and pop instructions that also decrement and increment the register respectively.
- Register indexed, register based, and register based indexed addressing are

used to access components of arrays such as the one in Figure 4-14, and components buried beneath the top of the stack, in a data structure known as the **stack frame**, which is discussed in the next section.

4.6 Subroutine Linkage and Stacks

A **subroutine**, sometimes called a **function** or **procedure**, is a sequence of instructions that is invoked in a manner that makes it appear to be a single instruction in a high level view. When a program calls a subroutine, control is passed from the program to the subroutine, which executes a sequence of instructions and then returns to the location just past where it was called. There are a number of methods for passing arguments to and from the called routine, referred to as **calling conventions**. The process of passing arguments between routines is referred to as **subroutine linkage**.

One calling convention simply places the arguments in registers. The code in Figure 4-15 shows a program that loads two arguments into %r1 and %r2, calls

! Calling routine	! Called routine
:	! %r3 ← %r1 + %r2
:	
ld [x], %r1	
ld [y], %r2	add_1: addcc %r1, %r2, %r3
call add_1	jmpl %r15 + 4, %r0
st %r3, [z]	
:	
:	
x: 53	
y: 10	
z: 0	

Figure 4-15 Subroutine linkage using registers.

subroutine `add_1`, and then retrieves the result from %r3. Subroutine `add_1` takes its operands from %r1 and %r2, and places the result in %r3 before returning via the `jmpl` instruction. This method is fast and simple, but it will not work if the number of arguments that are passed between the routines exceeds the number of free registers, or if subroutine calls are deeply nested.

A second calling convention creates a **data link area**. The address of the data link area is passed in a predetermined register to the called routine. Figure 4-16 shows an example of this method of subroutine linkage. The `.dwb` pseudo-op in the calling routine sets up a data link area that is three words long, at addresses `x`,

! Calling routine	! Called routine
⋮	! x[2] ← x[0] + x[1]
st %r1, [x]	add_2: ld %r5, %r8
st %r2, [x+4]	ld %r5 + 4, %r9
sethi x, %r5	addcc %r8, %r9, %r10
srl %r5, 10, %r5	st %r10, %r5 + 8
call add_2	jmp1 %r15 + 4, %r0
ld [x+8], %r3	
⋮	
! Data link area	
x: .dwb 3	

Figure 4-16 Subroutine linkage using a data link area.

$x+4$, and $x+8$. The calling routine loads its two arguments into x and $x+4$, calls subroutine `add_2`, and then retrieves the result passed back from `add_2` from memory location $x+8$. The address of data link area x is passed to `add_2` in register `%r5`.

Note that `sethi` must have a constant for its source operand, and so the assembler recognizes the `sethi` construct shown for the calling routine and replaces x with its address. The `srl` that follows the `sethi` moves the address x into the least significant 22 bits of `%r5`, since `sethi` places its operand into the leftmost 22 bits of the target register. An alternative approach to loading the address of x into `%r5` would be to use a storage location for the address of x , and then simply apply the `ld` instruction to load the address into `%r5`. While the latter approach is simpler, the `sethi/srl` approach is faster because it does not involve a time consuming access to the memory.

Subroutine `add_2` reads its two operands from the data link area at locations `%r5` and `%r5 + 4`, and places its result in the data link area at location `%r5 + 8` before returning. By using a data link area, arbitrarily large blocks of data can be passed between routines without copying more than a single register during subroutine linkage. Recursion can create a burdensome bookkeeping overhead, however, since a routine that calls itself will need several data link areas. Data link areas have the advantage that their size can be unlimited, but also have the disadvantage that the size of the data link area must be known at assembly time.

A third calling convention uses a stack. The general idea is that the calling routine pushes all of its arguments (or pointers to arguments, if the data objects are large) onto a last-in-first-out stack. The called routine then pops the passed argu-

ments from the stack, and pushes any return values onto the stack. The calling routine then retrieves the return value(s) from the stack and continues execution. A register in the CPU, known as the **stack pointer**, contains the address of the top of the stack. Many machines have `push` and `pop` instructions that automatically decrement and increment the stack pointer as data items are pushed and popped.

An advantage of using a stack is that its size grows and shrinks as needed. This supports arbitrarily deep nesting of procedure calls without having to declare the size of the stack at assembly time. An example of passing arguments using a stack is shown in Figure 4-17. Register `%r14` serves as the stack pointer (`%sp`) which is

! Calling routine	! Called routine
:	! Arguments are on stack.
:	! %sp[0] ← %sp[0] + %sp[4]
%sp .equ %r14	%sp .equ %r14
addcc %sp, -4, %sp	add_3: ld %sp, %r8
st %r1, %sp	addcc %sp, 4, %sp
addcc %sp, -4, %sp	ld %sp, %r9
st %r2, %sp	addcc %r8, %r9, %r10
call add_3	st %r10, %sp
ld %sp, %r3	jmp1 %r15 + 4, %r0
addcc %sp, 4, %sp	
:	
:	

Figure 4-17 Subroutine linkage using a stack.

initialized by the operating system prior to execution of the calling routine. The calling routine places its arguments (`%r1` and `%r2`) onto the stack by decrementing the stack pointer (which moves `%sp` to the next free word above the stack) and by storing each argument on the new top of the stack. Subroutine `add_3` is called, which pops its arguments from the stack, performs an addition operation, and then stores its return value on the top of the stack before returning. The calling routine then retrieves its argument from the top of the stack and continues execution.

For each of the calling conventions, the `call` instruction is used, which saves the current PC in `%r15`. When a subroutine finishes execution, it needs to return to the instruction that *follows* the call, which is one word (four bytes) past the saved PC. Thus, the statement “`jmp1 %r15 + 4, %r0`” completes the return. If the called routine calls *another* routine, however, then the value of the PC that was originally saved in `%r15` will be overwritten by the nested call, which means that

a correct return to the original calling routine through `%r15` will no longer be possible. In order to allow nested calls and returns, the current value of `%r15` (which is called the **link register**) should be saved on the stack, along with any other registers that need to be restored after the return.

If a register based calling convention is used, then the link register should be saved in one of the unused registers before a nested call is made. If a data link area is used, then there should be space reserved within it for the link register. If a stack scheme is used, then the link register should be saved on the stack. For each of the calling conventions, the link register and the local variables in the called routines should be saved before a nested call is made, otherwise, a nested call to the same routine will cause the local variables to be overwritten.

There are many variations to the basic calling conventions, but the stack-oriented approach to subroutine linkage is probably the most popular. When a stack based calling convention is used that handles nested subroutine calls, a **stack frame** is built that contains arguments that are passed to a called routine, the return address for the calling routine, and any local variables. A sample high level program is shown in Figure 4-18 that illustrates nested function calls. The operation that the program performs is not important, nor is the fact that the C programming language is used, but what is important is how the subroutine calls are implemented.

The behavior of the stack for this program is shown in Figure 4-19. The main program calls `func_1` with arguments 1 and 2, and then calls `func_2` with argument 10 before finishing execution. Function `func_1` has two local variables `i` and `j` that are used in computing the return value `j`. Function `func_2` has two local variables `m` and `n` that are used in creating the arguments to pass through to `func_1` before returning `m`.

The stack pointer (`%r14` by convention, which will be referred to as `%sp`) is initialized before the program starts executing, usually by the operating system. The compiler is responsible for implementing the calling convention, and so the compiler produces code for pushing parameters and the return address onto the stack, reserving room on the stack for local variables, and then reversing the process as routines return from their calls. The stack behavior shown in Figure 4-19 is thus produced as the result of executing compiler generated code, but the code may just as well have been written directly in assembly language.

As the main program begins execution, the stack pointer points to the top ele-

```

Line /* C program showing nested subroutine calls */
No.
00 main()
01 {
02     int w, z;          /* Local variables */
03     w = func_1(1,2);   /* Call subroutine func_1 */
04     z = func_2(10);    /* Call subroutine func_2 */
05 }                      /* End of main routine */

06 int func_1(x,y)       /* Compute x * x + y */
07 int x, y;             /* Parameters passed to func_1 */
08 {
09     int i, j;          /* Local variables */
10     i = x * x;
11     j = i + y;
12     return(j);        /* Return j to calling routine */
13 }

14 int func_2(a)          /* Compute a * a + a + 5 */
15 int a;                 /* Parameter passed to func_2 */
16 {
17     int m, n;          /* Local variables */
18     n = a + 5;
19     m = func_1(a,n);
20     return(m);         /* Return m to calling routine */
21 }

```

Figure 4-18 A C program illustrating nested function calls.

ment of the system stack (Figure 4-19a). When the main routine calls `func_1` at line 03 of the program shown in Figure 4-18 with arguments 1 and 2, the arguments are pushed onto the stack, as shown in Figure 4-19b. Control is then transferred to `func_1` through a `call` instruction (not shown), and `func_1` then saves the return address, which is in `%r15` as a result of the `call` instruction, onto the stack (Figure 4-19c). Stack space is reserved for local variables `i` and `j` of `func_1` (Figure 4-19d). At this point, we have a complete stack frame for the `func_1` call as shown in Figure 4-19d, which is composed of the arguments passed to `func_1`, the return address to the main routine, and the local variables for `func_1`.

Just prior to `func_1` returning to the calling routine, it releases the stack space for its local variables, retrieves the return address from the stack, releases the stack space for the arguments passed to it, and then pushes its return value onto the stack as shown in Figure 4-19e. Control is then returned to the calling routine through a `jmp1` instruction, and the calling routine is then responsible for retrieving the returned value from the stack and decrementing the stack pointer

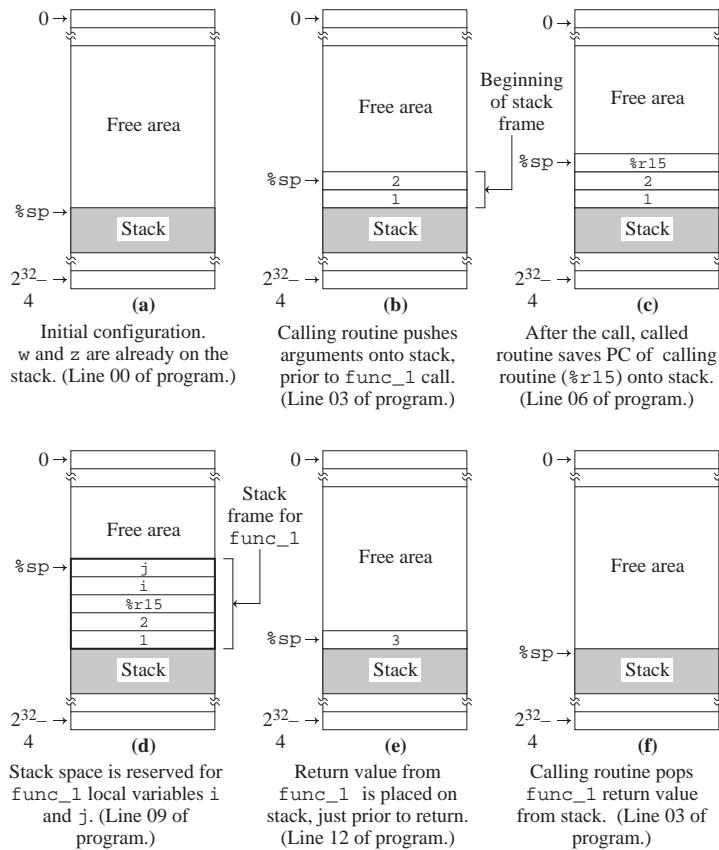


Figure 4-19 (a-f) Stack behavior during execution of the program shown in Figure 4-18.

to its position from before the call, as shown in Figure 4-19f. Routine `func_2` is then executed, and the process of building a stack frame starts all over again as shown in Figure 4-19g. Since `func_2` makes a call to `func_1` before it returns, there will be stack frames for both `func_2` and `func_1` on the stack at the same time as shown in Figure 4-19h. The process then unwinds as before, finally resulting in the stack pointer at its original position as shown in Figure 4-19(i-k).

4.7 Input and Output in Assembly Language

Finally, we come to ways in which an assembly language program can communicate with the outside world: input and output (I/O) activities. One way that communication between I/O devices and the rest of the machine can be handled is with special instructions, and with a special I/O bus reserved for this purpose. An alternative method for interacting with I/O devices is through the use of

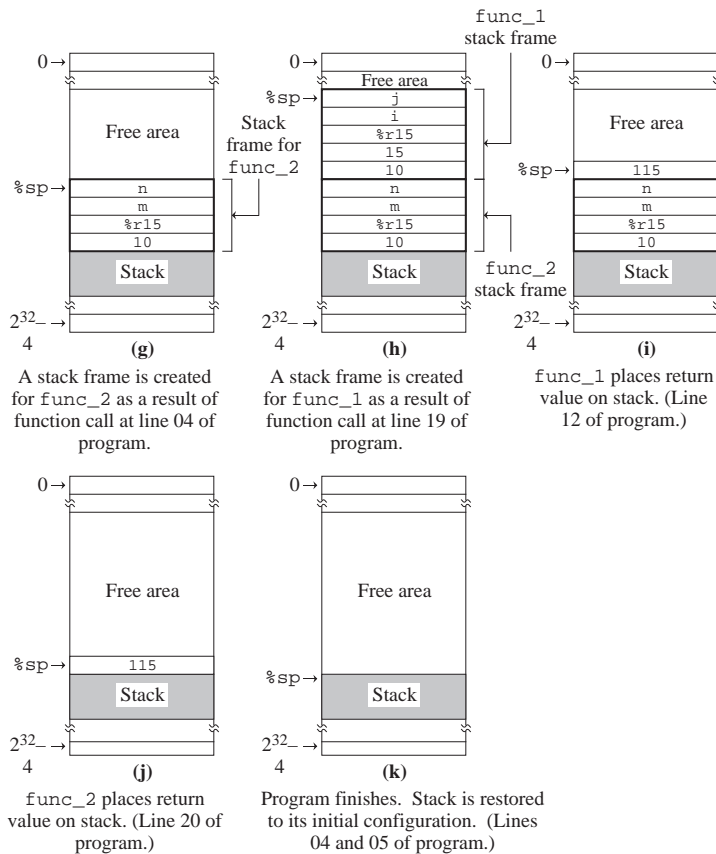


Figure 4-19 (g-k) (Continued.)

memory mapped I/O, in which devices occupy sections of the address space where no ordinary memory exists. Devices are accessed as if they are memory locations, and so there is no need for handling devices with new instructions.

As an example of memory mapped I/O, consider again the memory map for the ARC, which is illustrated in Figure 4-20. We see a few new regions of memory, for two add-in video memory modules and for a **touchscreen**. A touchscreen comes in two forms, photonic and electrical. An illustration of the photonic version is shown in Figure 4-21. A matrix of beams covers the screen in the horizontal and vertical dimensions. If the beams are interrupted (by a finger for example) then the position is determined by the interrupted beams. (In an alternative version of the touchscreen, the display is covered with a touch sensitive surface. The user must make contact with the screen in order to register a selection.)

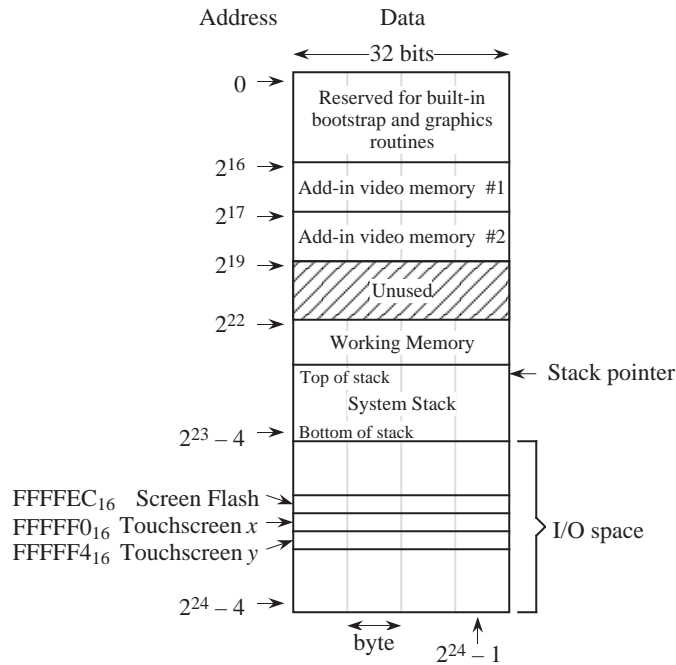


Figure 4-20 Memory map for the ARC, showing memory mapping.

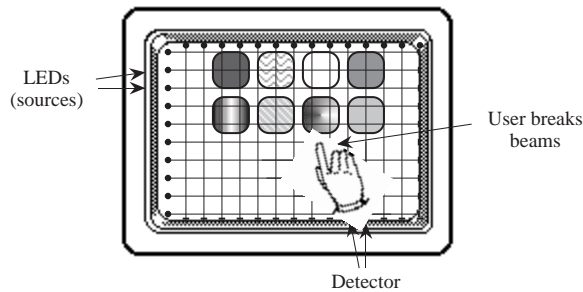


Figure 4-21 A user selecting an object on a touchscreen.

The only real memory occupies the address space between 2^{22} and $2^{23} - 1$. (Remember: $2^{23} - 4$ is the address of the leftmost byte of the highest word in the big-endian format.) The rest of the address space is occupied by other components. The address space between 0 and $2^{16} - 1$ (inclusive) contains built-in programs for the power-on bootstrap operation and basic graphics routines. The address space between 2^{16} and $2^{19} - 1$ is used for two add-in video memory modules, which we will study in Problem Figure 4.3. Note that valid informa-

tion is available only when the add-in memory modules are physically inserted into the machine.

Finally, the address space between 2^{23} and $2^{24} - 1$ is used for I/O devices. For this system, the X and Y coordinates that mark the position where a user has made a selection are automatically updated in registers that are placed in the memory map. The registers are accessed by simply reading from the memory locations where these registers are located. The “Screen Flash” location causes the screen to flash whenever it is written.

Suppose that we would like to write a simple program that flashes the screen whenever the user changes position. The flowchart in Figure 4-22 illustrates how

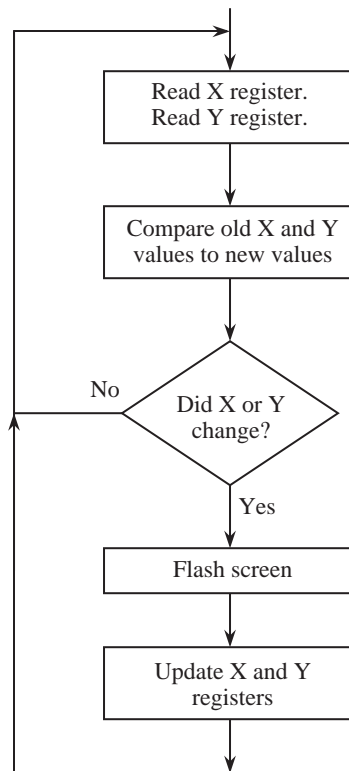


Figure 4-22 Flowchart illustrating the control structure of a program that tracks a touchscreen.

this might be done. The X and Y registers are first read, and are then compared with the previous X and Y values. If either position has changed, then the screen is flashed and the previous X and Y values are updated and the process repeats. If

neither position has changed, then the process simply repeats. This is an example of the programmed I/O method of accessing a device. (See problem 4.3 at the end of the chapter for a more detailed description.)

4.8 Case Study: The Java Virtual Machine ISA

Java is a high-level programming language developed by Sun Microsystems that has taken a prominent position in the programming community. A key aspect of Java is that Java binary codes are platform-independent, which means that the same compiled code can run without modification on any computer that supports the **Java Virtual Machine** (JVM). The JVM is how Java achieves its platform-independence: a standard specification of the JVM is implemented in the native instruction sets of many underlying machines, and compiled Java codes can then run in any JVM environment.

Programs that are written in fully compiled languages like C, C++, and Fortran, are compiled into the native code of the target architecture, and are generally not portable across platforms unless the source code is recompiled for the target machine. Interpreted languages, like Perl, Tcl, AppleScript, and shell script, are largely platform independent, but can execute 100 to 200 times slower than a fully compiled language. Java programs are compiled into an intermediate form known as **bytecodes**, which execute on the order of 10 times more slowly than fully compiled languages, but the cross-platform compatibility and other language features make Java a favorable programming language for many applications.

A high level view of the JVM architecture is shown in Figure 4-23. The JVM is a stack-based machine, which means that the operands are pushed and popped from a stack, instead of being transferred among general purpose registers. There are, however, a number of special purpose registers, and also a number of local variables that serve the function of general purpose registers in a “real” (non-virtual) architecture. The Java Execution Engine takes compiled Java bytecodes at its input, and interprets the bytecodes in a software implementation of the JVM, or executes the bytecodes directly in a hardware implementation of the JVM.

Figure 4-24 shows a Java implementation of the SPARC program we studied in Figure 4-13. The figure shows both the Java source program and the bytecodes into which it was compiled. The bytecode file is known as a Java **class file** (which is what a compiled Java program is called.)

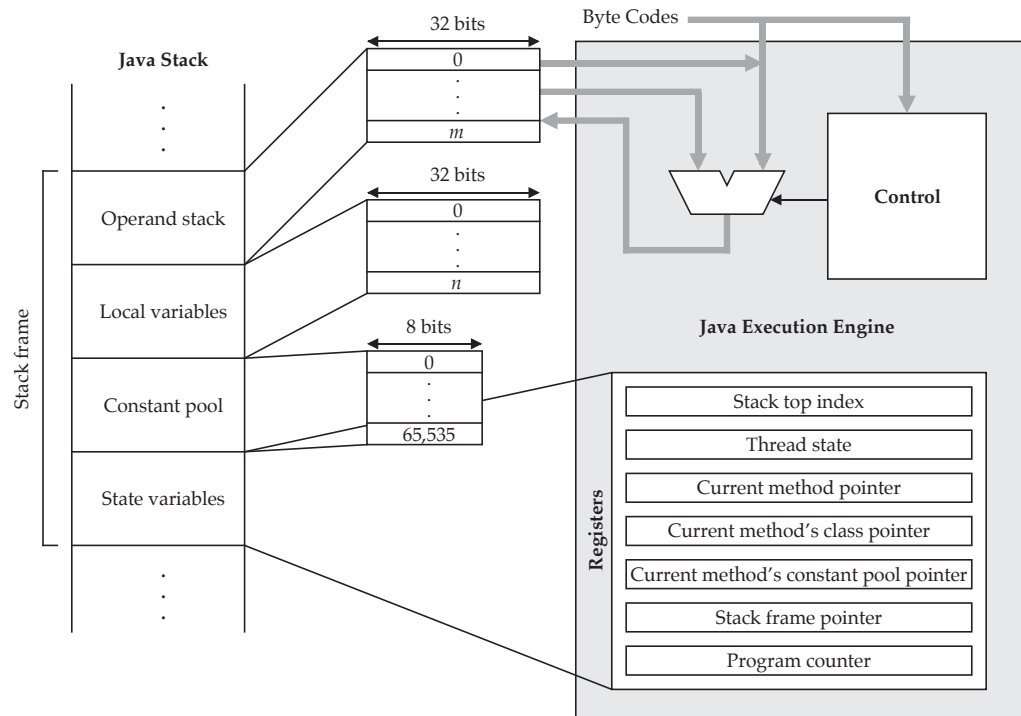


Figure 4-23 Architecture of the Java virtual machine.

Only a small number of bytes in a class file actually contain instructions; the rest is overhead that the file must contain in order to run on the JVM. In Figure 4-25 we have “disassembled” the bytecodes back to their higher-level format. The bytecode locations are given in hexadecimal, starting at location 0x00. The first 4 bytes contain the magic number 0xcafebabe which identifies the program as a compiled Java class file. The major version and minor version numbers refer to the Java runtime system for which the program is compiled. The number of entries in the **constant pool** follows, which is actually 17 in this example: the first entry (constant pool location 0) is always reserved for the JVM, and is not included in the class file, although indexing into the constant pool starts at location 0 as if it is explicitly represented. The constant pool contains the names of **methods** (functions), attributes, and other information used by the runtime system.

The remainder of the file is mostly composed of the constant pool, and executable Java instructions. We will not cover all details of the Java class file here. The reader is referred to (Meyer & Downing, 1997) for a full description of the Java

```
// This is file add.java

public class add {
    public static void main(String args[]) {
        int x=15, y=9, z=0;
        z = x + y;
    }
}

0000 cafe babe 0003 002d 0012 0700 0e07 0010 .....
0010 0a00 0200 040c 0007 0005 0100 0328 2956 .....()V
0020 0100 1628 5b4c 6a61 7661 2f6c 616e 672f ...([Ljava/lang/
0030 5374 7269 6e67 3b29 5601 0006 3c69 6e69 String;)V...<ini
0040 743e 0100 0443 6f64 6501 000d 436f 6e73 t>...Code...Cons
0050 7461 6e74 5661 6c75 6501 000a 4578 6365 tantValue...Exce
0060 7074 696f 6e73 0100 0f4c 696e 654e 756d ptions...LineNum
0070 6265 7254 6162 6c65 0100 0e4c 6f63 616c berTable...Local
0080 5661 7269 6162 6c65 7301 000a 536f 7572 Variables...Sour
0090 6365 4669 6c65 0100 0361 6464 0100 0861 ceFile...add...a
00a0 6464 2e6a 6176 6101 0010 6a61 7661 2f6c dd.java...java/l
00b0 616e 672f 4f62 6a65 6374 0100 046d 6169 ang/Object...mai
00c0 6e00 2100 0100 0200 0000 0000 0200 0900 n.....
00d0 1100 0600 0100 0800 0000 2d00 0200 0400 .....
00e0 0000 0d10 0f3c 1009 3d03 3e1b 1c60 3eb1 .....
00f0 0000 0001 000b 0000 000e 0003 0000 0004 .....
0100 0008 0006 000c 0002 0001 0007 0005 0001 .....
0110 0008 0000 001d 0001 0001 0000 0005 2ab7 .....
0120 0003 b100 0000 0100 0b00 0000 0600 0100 .....
0130 0000 0100 0100 0d00 0000 0200 0f00 .....
```

Figure 4-24 Java program and compiled class file.

class file format.

The actual code that corresponds to the Java source program, which simply adds the constants 15 and 9, and returns the result (24) to the calling routine on the stack, appears in locations 0x00e3 - 0x00ef. Figure 4-26 shows how that portion of the bytecode is interpreted. The program pushes the constants 15 and 9 onto the stack, using local variables 0 and 1 as intermediaries, and invokes the `iadd` instruction which pops the top two stack elements, adds them, and places the result on the top of the stack. The program then returns.

A cursory glance at the code shows some of the reasons why the JVM runs 10 times slower than native code. Notice that the program stores the arguments in local variables 1 and 2, and then transfers them to the Java stack before adding them. This transfer would be viewed as redundant by native code compilers for other languages, and would be eliminated. Given this example alone, there is probably considerable room for speed improvements from the 10× slower execution time of today's JVMs. Other improvements may also come in the form of **just in time** (JIT) compilers. Rather than interpreting the JVM bytecodes one

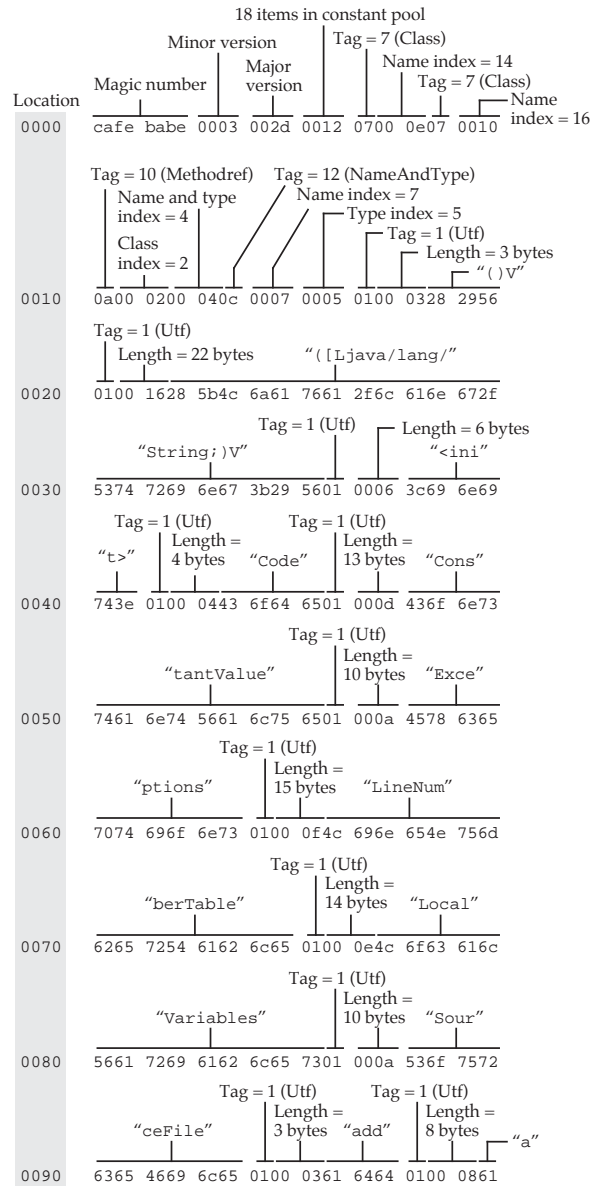


Figure 4-25 A Java class file.

by one into the target machine code each time they are encountered, JIT compilers take advantage of the fact that most programs spend most of their time in loops and other iterative routines. As the JIT encounters each line of code for the first time, it compiles it into native code and stores it away in memory for possi-

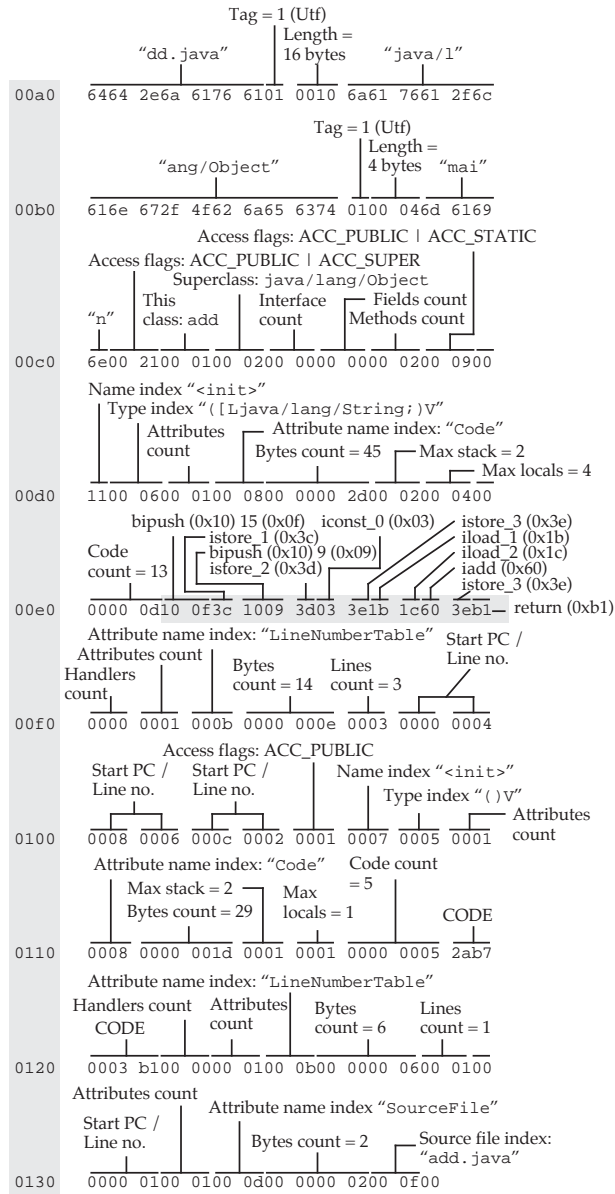


Figure 4-25 (A Java class file (Continued)).

ble later use. The next time that code is executed, it is the native, compiled form that is executed rather than the bytecodes.

Location	Code	Mnemonic	Meaning
0x00e3	0x10	bipush	Push next byte onto stack
0x00e4	0x0f	15	Argument to bipush
0x00e5	0x3c	istore_1	Pop stack to local variable 1
0x00e6	0x10	bipush	Push next byte onto stack
0x00e7	0x09	9	Argument to bipush
0x00e8	0x3d	istore_2	Pop stack to local variable 2
0x00e9	0x03	iconst_0	Push 0 onto stack
0x00ea	0x3e	istore_3	Pop stack to local variable 3
0x00eb	0x1b	iload_1	Push local variable 1 onto stack
0x00ec	0x1c	iload_2	Push local variable 2 onto stack
0x00ed	0x60	iadd	Add top two stack elements
0x00ef	0xb1	return	Return

Figure 4-26 Disassembled version of the code that implement the Java program in Figure 4-24.

■ SUMMARY

In this chapter, we introduced the ARC ISA, and studied some general properties of ISAs. In the design of an instruction set, a balance must be struck between system performance and the characteristics of the technology in which the processor is implemented. Interaction between the CPU and the memory is a key consideration.

When a memory access is made, the way in which the address is calculated is called the memory addressing mode. We examined the sequence of computations that can be combined to make up an addressing mode. We also looked at some specific cases which are commonly identified by name.

We also looked at several parts of a computer system that play a role in the execution of a program. We learned that programs are made up of sequences of instructions, which are taken from the instruction set of the CPU. In the next chapter, we will study how these sequences of instructions are translated into object code.

■ FURTHER READING

The material in this chapter is for the most part a collection of the historical experience gained in fifty years of stored program computer designs. Although

each generation of computer systems is typically identified by a specific hardware technology, there have also been historically important instruction set architectures. In the first generation systems of the 1950's, such as Von Neuman's EDVAC, Eckert and Mauchly's UNIVAC and the IBM 701, programming was performed by hand in machine language. Although simple, these instruction set architectures defined the fundamental concepts surrounding opcodes and operands.

The concept of an instruction set architecture as an identifiable entity can be traced to the designers of the IBM S/360 in the 1960's. The VAX architecture for Digital Equipment Corporation can also trace its roots to this period when the minicomputers, the PDP-4 and PDP-8 were being developed. Both the S/360 and VAX are two-address architectures. Significant one-address architectures include the Intel 8080 which is the predecessor to the modern 80x86, and its contemporary at that time: the Zilog Z-80. As a zero-address architecture, the Burroughs B5000 is also of historical significance.

There are a host of references that cover the various machine languages in existence, too many to enumerate here, and so we mention only a few of the more celebrated cases. The machine languages of Babbage's machines are covered in (Bromley, 1987). The machine language of the early Institute for Advanced Study (IAS) computer is covered in (Stallings, 1996). The IBM 360 machine language is covered in (Strubl, 1975). The machine language of the 68000 can be found in (Gill, 1987) and the machine language of the SPARC can be found in (SPARC, 1992). A full description of the JVM and the Java class file format can be found in (Meyer & Downing, 1997.)

Bromley, A. G., "The Evolution of Babbage's Calculating Engines," *Annals of the History of Computing*, **9**, pp. 113-138, (1987).

Gill, A., E. Corwin, and A. Logar, *Assembly Language Programming for the 68000*, Prentice-Hall, Englewood Cliffs, New Jersey, (1987).

Meyer, J. and T. Downing, *Java Virtual Machine*, O'Reilly & Associates, Sebastopol, California, (1997).

SPARC International, Inc., *The SPARC Architecture Manual: Version 8*, Prentice Hall, Englewood Cliffs, New Jersey, (1992).

Stallings, W., *Computer Organization and Architecture*, 4/e, Prentice Hall, Upper

Saddle River, (1996).

Struble, G. W., *Assembler Language Programming: The IBM System/360 and 370*, 2/e, Addison-Wesley, Reading, (1975).

■ PROBLEMS

- 4.1** A memory has 2^{24} addressable locations. What is the smallest width in bits that the address can be while still being able to address all 2^{24} locations?
- 4.2** What are the lowest and highest addresses in a 2^{20} byte memory, in which a four-byte word is the smallest addressable unit?
- 4.3** The memory map for the ARC is shown in Figure 4-20.

(a) How much memory (in bytes) is available for each of the add-in video memory modules? (Give your answer as powers of two or sums of powers of two, *e.g.* 2^{10} .)

(b) When a finger is drawn across the touchscreen, the horizontal (x) and vertical (y) positions of the joystick are updated in registers that are accessed at locations $(\text{FFFFF0})_{16}$ and $(\text{FFFFF4})_{16}$, respectively. When the number '1' is written to the register at memory location $(\text{FFFFEC})_{16}$ the screen flashes, and then location $(\text{FFFFEC})_{16}$ is automatically cleared to zero by the hardware (the software does not have to clear it). Write an ARC program that flashes the screen every time the user's position changes. Use the skeleton program shown below.

```
.begin
ld    [x], %r7! %r7 and %r8 now point to the
ld    [y], %r8!   touchscreen x and y locations
ld    [flash], %r9! %r9 points to flash location
loop:ld    %r7, %r1 ! Load current touchscreen position
ld    %r8, %r2!   in %r1=x and %r2=y
ld    [old_x], %r3! Load old touchscreen position
ld    [old_y], %r4!   in %r3=x and %r4=y
orncc %r3, %r0, %r3! Form 1's complement of old_x
addcc %r3, 1, %r3! Form 2's complement of old_x
addcc %r1, %r3, %r3! %r3 <- x - old_x
be    x_not_moved! Branch if x did not change
ba    moved      ! x changed, so no need to check y
```



```

x_not_moved:    ! Your code starts here, about four lines.

                <-- YOUR CODE GOES HERE

                ! This portion of the code is entered only if joystick
                ! is moved.
                ! Flash screen; store new x, y values; repeat.
moved:orcc  %r0, 1, %r5! Place 1 in %r5
          st   %r5, %r9! Store 1 in flash register
          st   %r1, [old_x]! Update old joystick position
          st   %r2, [old_y]!   with current position
          ba   loop! Repeat
flash:     #FFFFEC! Location of flash register
x:         #FFFFFF0    ! Location of touchscreen x register
y:         #FFFFFF4    ! Location of touchscreen y register
old_x:     0          ! Previous x position
old_y:     0          ! Previous y position
          .end

```

4.4 Write an ARC subroutine that performs a swap operation on the 32-bit operands $x = 25$ and $y = 50$, which are stored in memory. Use as few registers as you can.

4.5 A section of ARC assembly code is shown below. What does it do? Express your answer in terms of the actions it goes through. Does it add up numbers, or clear something out? Does it simulate a **for** loop, a **while** loop, or something else? Assume that a and b are memory locations that are defined elsewhere in the code.

```

Y:      ld     [k], %r1
        addcc %r1, -4, %r1
        st     %r1, [k]
        bneg  X
        ld     [a], %r1, %r2
        ld     [b], %r1, %r3
        addcc %r2, %r3, %r4
        st     %r4, %r1, [c]
        ba     Y
X:      jmp1   %r15 + 4, %r0
k:      40

```

4.6 A pocket pager contains a small processor with 2^7 8-bit words of memory.

The ISA has four registers: R0, R1, R2, and R3. The instruction set is shown in Figure 4-27, as well as the bit patterns that correspond to each register, the

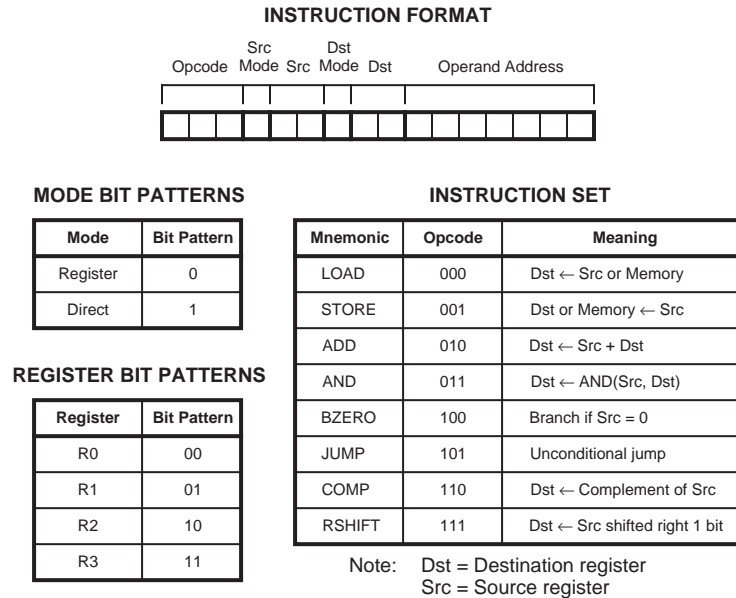


Figure 4-27 A pocket pager ISA.

instruction format, and the **modes**, which determine if the operand is a register (mode bit = 0) or the operand is a memory location (mode bit = 1). Either or both of the operands can be registers, but both operands cannot be memory locations. If the source or destination is a memory location, then the corresponding source or destination field in the instruction is not used since the address field is used instead.

(a) Write a program using object code (not mnemonics) that swaps the contents of registers R0 and R1. You are free to use the other registers as necessary, but do not use memory. Use no more than four lines of code (fewer lines are possible). Place 0's in any positions where the value does not matter.

(b) Write a program using object code that swaps the contents of memory locations 12 and 13. As in part (a), you are free to use the other registers as necessary, but do not use other memory locations. Place 0's in any positions where the value does not matter.

4.7 An ARC program calls the subroutine `foo`, passing it three arguments, `a`, `b`, and `c`. The subroutine has two local variables, `m` and `n`. Show the position of the stack pointer and the contents of the relevant stack elements for a stack based calling convention at the points in the program shown below:

(1) just before executing the `call` at label `x`;

(2) when the stack frame for `foo` is completed;

(3) just before executing the `ld` at label `z` (*i.e.*, when the calling routine resumes).

Use the stack notation shown in Figure 4-19.

```

! Push the arguments a, b, and c
x:      call  foo
z:      ld    %r1, %r2
        .
        .
        .
foo: ! Subroutine starts here
        .
        .
        .
y:      jmp1  %r15 + 4, %r0

```

4.8 Why does `sethi` only load the high 22 bits of a register? It would be more useful if `sethi` loaded all 32 bits of a register. What is the problem with having `sethi` load all 32 bits?

4.9 Which of the three subroutine linkage conventions covered in this chapter (registers, data link area, stack) is used in Figure 4-14?

4.10 A program compiled for a SPARC ISA writes the 32-bit unsigned integer `0xABCDEF01` to a file, and reads it back correctly. The same program compiled for a Pentium ISA also works correctly. However, when the file is transferred between machines, the program incorrectly reads the integer from the file as `0x01EFCDA B`. What is going wrong?

4.11 Refer to Figure 4-25. Show the Java assembly language instructions for the

code shown in locations 0x011e - 0x0122. Use the syntax format shown in locations 0x00e3 - 0x00ef of that same figure.

You will need to make use of the following Java instructions:

`invokespecial n` (opcode 0xb7) – Invoke a method with index `n` into the constant pool. Note that `n` is a 16-bit (two-byte) index that follows the `invokespecial` opcode.

`aload_0` (opcode 0x2a) – Push local variable 0 onto the stack.

4.12 Is the JVM a little-endian or big-endian machine? Hint: Examine the first line of the bytecode program in Figure 4-24.

4.13 Write an ARC program that implements the bytecode program shown in Figure 4-26. Assume that, analogous in the code in the figure, the arguments are passed on a stack, and that the return value is placed on the top of the stack.

4.14 A JVM is implemented using the ARC ISA.

a) How much memory traffic will be generated when the program of Figure 4-26 executes?

b) For exercise 4-13, compute the memory traffic your program will generate. Then, for part (a) above, compare that traffic with the amount generated by your program. If most of the execution time of a program is due to its memory accesses, how much faster will your program be compared to the program in Figure 4-26?

4.15 Can a Java bytecode program ever run as fast as a program written in the native language of the processor? Defend your answer in one or two paragraphs.

4.16 (a) Write three-address, two-address, and one-address programs to compute the function $A = (B-C)*(D-E)$. Assume 8-bit opcodes, 16-bit operands and addresses, and that data is moved to and from memory in 16-bit chunks. (Also assume that the opcode must be transferred from memory by itself.) Your code should not overwrite any of the operands. Use any temporary regis-

ters needed.

b. Compute the size of your program in bytes.

c. Compute the memory traffic your program will generate at execution time, including instruction fetches.

4.17 Repeat Exercise 4.12 above, using ARC assembly language.

