Chapter 4

Processor Architecture

Modern microprocessors are among the most complex systems ever created by humans. A single silicon chip, roughly the size of a fingernail, can contain a complete high-performance processor, large cache memories, and the logic required to interface it to external devices. In terms of performance, the processors implemented on a single chip today dwarf the room-sized supercomputers that cost over $10 million just 20 years ago. Even the embedded processors found in everyday appliances such as cell phones, personal digital assistants, and handheld game systems are far more powerful than the early developers of computers ever envisioned.

Thus far, we have only viewed computer systems down to the level of machine-language programs. We have seen that a processor must execute a sequence of instructions, where each instruction performs some primitive operation, such as adding two numbers. An instruction is encoded in binary form as a sequence of 1 or more bytes. The instructions supported by a particular processor and their byte-level encodings are known as its *instruction-set architecture* (ISA). Different "families" of processors, such as Intel IA32, IBM/Freescale PowerPC, and the ARM processor family have different ISAs. A program compiled for one type of machine will not run on another. On the other hand, there are many different models of processors within a single family. Each manufacturer produces processors of ever-growing performance and complexity, but the different models remain compatible at the ISA level. Popular families, such as IA32, have processors supplied by multiple manufacturers. Thus, the ISA provides a conceptual layer of abstraction between compiler writers, who need only know what instructions are permitted and how they are encoded, and processor designers, who must build machines that execute those instructions.

In this chapter, we take a brief look at the design of processor hardware. We study the way a hardware system can execute the instructions of a particular ISA. This view will give you a better understanding of how computers work and the technological challenges faced by computer manufacturers. One important concept is that the actual way a modern processor operates can be quite different from the model of computation implied by the ISA. The ISA model would seem to imply *sequential* instruction execution, where each instruction is fetched and executed to completion before the next one begins. By executing different parts of multiple instructions simultaneously, the processor can achieve higher performance than if it executed just one instruction at a time. Special mechanisms are used to make sure the processor computes the same results as it would with sequential execution. This idea of using clever tricks to improve performance while maintaining the functionality of a simpler and more abstract model is well known in computer science.

Examples include the use of caching in Web browsers and information retrieval data structures such as balanced binary trees and hash tables.

Chances are you will never design your own processor. This is a task for experts working at fewer than 100 companies worldwide. Why, then, should you learn about processor design?

- *It is intellectually interesting and important.* There is an intrinsic value in learning how things work. It is especially interesting to learn the inner workings of a system that is such a part of the daily lives of computer scientists and engineers and yet remains a mystery to many. Processor design embodies many of the principles of good engineering practice. It requires creating a simple and regular structure to perform a complex task.
- *Understanding how the processor works aids in understanding how the overall computer system works.* In Chapter 6, we will look at the memory system and the techniques used to create an image of a very large memory with a very fast access time. Seeing the processor side of the processor-memory interface will make this presentation more complete.
- *Although few people design processors, many design hardware systems that contain processors.* This has become commonplace as processors are embedded into real-world systems such as automobiles and appliances. Embedded-system designers must understand how processors work, because these systems are generally designed and programmed at a lower level of abstraction than is the case for desktop systems.
- *You just might work on a processor design.* Although the number of companies producing microprocessors is small, the design teams working on those processors are already large and growing. There can be over 1000 people involved in the different aspects of a major processor design.

In this chapter, we start by defining a simple instruction set that we use as a running example for our processor implementations. We call this the "Y86" instruction set, because it was inspired by the IA32 instruction set, which is colloquially referred to as "x86." Compared with IA32, the Y86 instruction set has fewer data types, instructions, and addressing modes. It also has a simpler byte-level encoding. Still, it is sufficiently complete to allow us to write simple programs manipulating integer data. Designing a processor to implement Y86 requires us to face many of the challenges faced by processor designers.

We then provide some background on digital hardware design. We describe the basic building blocks used in a processor and how they are connected together and operated. This presentation builds on our discussion of Boolean algebra and bit-level operations from Chapter 2. We also introduce a simple language, HCL (for "Hardware Control Language"), to describe the control portions of hardware systems. We will later use this language to describe our processor designs. Even if you already have some background in logic design, read this section to understand our particular notation.

As a first step in designing a processor, we present a functionally correct, but somewhat impractical, Y86 processor based on *sequential* operation. This processor executes a complete Y86 instruction on every clock cycle. The clock must run slowly enough to allow an entire series of actions to complete within one cycle. Such a processor could be implemented, but its performance would be well below what could be achieved for this much hardware.

With the sequential design as a basis, we then apply a series of transformations to create a *pipelined* processor. This processor breaks the execution of each instruction into five steps, each of which is handled

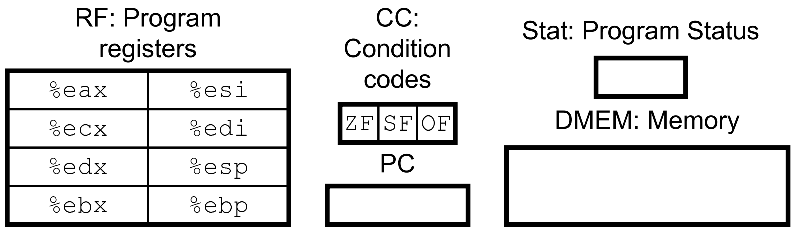

Figure 4.1: **Y86 programmer-visible state.** As with IA32, programs for Y86 access and modify the program registers, the condition code, the program counter (PC), and the memory. The status code indicates whether the program is running normally, or some special event has occurred.

by a separate section or *stage* of the hardware. Instructions progress through the stages of the pipeline, with one instruction entering the pipeline on each clock cycle. As a result, the processor can be executing the different steps of up to five instructions simultaneously. Making this processor preserve the sequential behavior of the Y86 ISA requires handling a variety of *hazard* conditions, where the location or operands of one instruction depend on those of other instructions that are still in the pipeline.

We have devised a variety of tools for studying and experimenting with our processor designs. These include an assembler for Y86, a simulator for running Y86 programs on your machine, and simulators for two sequential and one pipelined processor design. The control logic for these designs is described by files in HCL notation. By editing these files and recompiling the simulator, you can alter and extend the simulator's behavior. A number of exercises are provided that involve implementing new instructions and modifying how the machine processes instructions. Testing code is provided to help you evaluate the correctness of your modifications. These exercises will greatly aid your understanding of the material and will give you an appreciation for the many different design alternatives faced by processor designers.

Web Aside ARCH:VLOG presents a representation of our pipelined Y86 processor in the Verilog hardware description language. This involves creating modules for the basic hardware building blocks and for the overall processor structure. We automatically translate the HCL description of the control logic into Verilog. By first debugging the HCL description with our simulators, we eliminate many of the tricky bugs that would otherwise show up in the hardware design. Given a Verilog description, there are commercial and open-source tools to support simulation and *logic synthesis*, generating actual circuit designs for the microprocessors. So, although much of the effort we expend here is to create pictorial and textual descriptions of a system, much as one would when writing software, the fact that these designs can be automatically synthesized demonstrates that we are indeed creating a system that can be realized as hardware.

4.1 The Y86 Instruction Set Architecture

Defining an instruction set architecture, such as Y86, includes defining the different state elements, the set of instructions and their encodings, a set of programming conventions, and the handling of exceptional events.

4.1.1 Programmer-Visible State

As Figure 4.1 illustrates, each instruction in a Y86 program can read and modify some part of the processor state. This is referred to as the *programmer-visible* state, where the "programmer" in this case is either someone writing programs in assembly code or a compiler generating machine-level code. We will see in our processor implementations that we do not need to represent and organize this state in exactly the manner implied by the ISA, as long as we can make sure that machine-level programs appear to have access to the programmer-visible state. The state for Y86 is similar to that for IA32. There are eight *program registers*: `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%esp`, and `%ebp`. Each of these stores a word. Register `%esp` is used as a stack pointer by the push, pop, call, and return instructions. Otherwise, the registers have no fixed meanings or values. There are three single-bit *condition codes*, `ZF`, `SF`, and `OF`, storing information about the effect of the most recent arithmetic or logical instruction. The program counter (PC) holds the address of the instruction currently being executed.

The *memory* is conceptually a large array of bytes, holding both program and data. Y86 programs reference memory locations using *virtual addresses*. A combination of hardware and operating system software translates these into the actual, or *physical*, addresses indicating where the values are actually stored in memory. We will study virtual memory in more detail in Chapter 9. For now, we can think of the virtual memory system as providing Y86 programs with an image of a monolithic byte array.

A final part of the program state is a status code Stat, indicating the overall state of program execution. It will indicate either normal operation, or that some sort of *exception* has occurred, such as when an instruction attempts to read from an invalid memory address. The possible status codes and the handling of exceptions is described in Section 4.1.4.

4.1.2 Y86 Instructions

Figure 4.2 gives a concise description of the individual instructions in the Y86 ISA. We use this instruction set as a target for our processor implementations. The set of Y86 instructions is largely a subset of the IA32 instruction set. It includes only 4-byte integer operations, has fewer addressing modes, and includes a smaller set of operations. Since we only use 4-byte data, we can refer to these as "words" without any ambiguity. In this figure, we show the assembly-code representation of the instructions on the left and the byte encodings on the right. The assembly-code format is similar to the ATT format for IA32.

Here are some further details about the different Y86 instructions.

- The IA32 `movl` instruction is split into four different instructions: `irmovl`, `rrmovl`, `mrmovl`, and `rmmovl`, explicitly indicating the form of the source and destination. The source is either immediate (`i`), register (`r`), or memory (`m`). It is designated by the first character in the instruction name. The destination is either register (`r`) or memory (`m`). It is designated by the second character in the instruction name. Explicitly identifying the four types of data transfer will prove helpful when we decide how to implement them.

  The memory references for the two memory movement instructions have a simple base and displacement format. We do not support the second index register or any scaling of a register's value in the address computation.

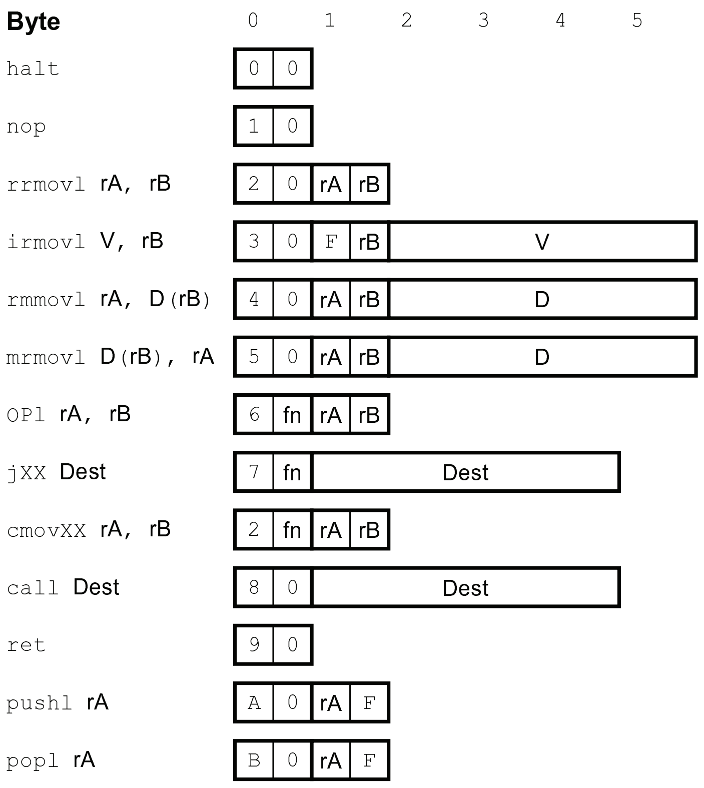

Figure 4.2: **Y86 instruction set.** Instruction encodings range between 1 and 6 bytes. An instruction consists of a 1-byte instruction specifier, possibly a 1-byte register specifier, and possibly a 4-byte constant word. Field `fn` specifies a particular integer operation (`OPl`), data movement condition (`cmovXX`), or branch condition (`jXX`). All numeric values are shown in hexadecimal.

Operations		Branches				Moves			
addl	6 0	jmp	7 0	jne	7 4	rrmovl	2 0	cmovne	2 4
subl	6 1	jle	7 1	jge	7 5	cmovle	2 1	cmovge	2 5
andl	6 2	jl	7 2	jg	7 6	cmovl	2 2	cmovg	2 6
xorl	6 3	je	7 3			cmove	2 3		

Figure 4.3: **Function codes for Y86 instruction set.** The code specifies a particular integer operation, branch condition, or data transfer condition. These instructions are shown as `OPl`, `jXX`, and `cmovXX` in Figure 4.2.

As with IA32, we do not allow direct transfers from one memory location to another. In addition, we do not allow a transfer of immediate data to memory.

- There are four integer operation instructions, shown in Figure 4.2 as `OPl`. These are `addl`, `subl`, `andl`, and `xorl`. They operate only on register data, whereas IA32 also allows operations on memory data. These instructions set the three condition codes `ZF`, `SF`, and `OF` (zero, sign, and overflow).
- The seven jump instructions (shown in Figure 4.2 as `jXX`) are `jmp`, `jle`, `jl`, `je`, `jne`, `jge`, and `jg`. Branches are taken according to the type of branch and the settings of the condition codes. The branch conditions are the same as with IA32 (Figure 3.12).
- There are six conditional move instructions (shown in Figure 4.2 as `cmovXX`): `cmovle`, `cmovl`, `cmove`, `cmovne`, `cmovge`, and `cmovg`. These have the same format as the register-register move instruction `rrmovl`, but the destination register is updated only if the condition codes satisfy the required constraints.
- The `call` instruction pushes the return address on the stack and jumps to the destination address. The `ret` instruction returns from such a call.
- The `pushl` and `popl` instructions implement push and pop, just as they do in IA32.
- The `halt` instruction stops instruction execution. IA32 has a comparable instruction, called `hlt`. IA32 application programs are not permitted to use this instruction, since it causes the entire system to suspend operation. For Y86, executing the `halt` instruction causes the processor to stop, with the status code set to `HLT`. (See Section 4.1.4.)

4.1.3 Instruction Encoding

Figure 4.2 also shows the byte-level encoding of the instructions. Each instruction requires between 1 and 6 bytes, depending on which fields are required. Every instruction has an initial byte identifying the instruction type. This byte is split into two 4-bit parts: the high-order, or *code*, part, and the low-order, or *function*, part. As you can see in Figure 4.2, code values range from `0` to `0xB`. The function values are significant only for the cases where a group of related instructions share a common code. These are given in Figure 4.3, showing the specific encodings of the integer operation, conditional move, and branch instructions. Observe that `rrmovl` has the same instruction code as the conditional moves. It can be viewed as an "unconditional move" just as the `jmp` instruction is an unconditional jump, both having function code `0`.

As shown in Figure 4.4, each of the eight program registers has an associated *register identifier* (ID) ranging from 0 to 7. The numbering of registers in Y86 matches what is used in IA32. The program registers are stored within the CPU in a *register file*, a small random-access memory where the register IDs serve as addresses. ID value `0xF` is used in the instruction encodings and within our hardware designs when we need to indicate that no register should be accessed.

Some instructions are just 1 byte long, but those that require operands have longer encodings. First, there can be an additional *register specifier byte*, specifying either one or two registers. These register fields are called rA and rB in Figure 4.2. As the assembly-code versions of the instructions show, they can specify the registers used for data sources and destinations, as well as the base register used in an address computation,

Number	Register name
0	%eax
1	%ecx
2	%edx
3	%ebx
4	%esp
5	%ebp
6	%esi
7	%edi
F	No register

Figure 4.4: **Y86 program register identifiers.** Each of the eight program registers has an associated identifier (ID) ranging from 0 to 7. ID 0xF in a register field of an instruction indicates the absence of a register operand.

depending on the instruction type. Instructions that have no register operands, such as branches and call, do not have a register specifier byte. Those that require just one register operand (irmovl, pushl, and popl) have the other register specifier set to value 0xF. This convention will prove useful in our processor implementation.

Some instructions require an additional 4-byte *constant word*. This word can serve as the immediate data for irmovl, the displacement for rmmovl and mrmovl address specifiers, and the destination of branches and calls. Note that branch and call destinations are given as absolute addresses, rather than using the PC-relative addressing seen in IA32. Processors use PC-relative addressing to give more compact encodings of branch instructions and to allow code to be copied from one part of memory to another without the need to update all of the branch target addresses. Since we are more concerned with simplicity in our presentation, we use absolute addressing. As with IA32, all integers have a little-endian encoding. When the instruction is written in disassembled form, these bytes appear in reverse order.

As an example, let us generate the byte encoding of the instruction rmmovl %esp,0x12345(%edx) in hexadecimal. From Figure 4.2, we can see that rmmovl has initial byte 40. We can also see that source register %esp should be encoded in the rA field, and base register %edx should be encoded in the rB field. Using the register numbers in Figure 4.4, we get a register specifier byte of 42. Finally, the displacement is encoded in the 4-byte constant word. We first pad 0x12345 with leading zeros to fill out 4 bytes, giving a byte sequence of 00 01 23 45. We write this in byte-reversed order as 45 23 01 00. Combining these, we get an instruction encoding of 404245230100.

One important property of any instruction set is that the byte encodings must have a unique interpretation. An arbitrary sequence of bytes either encodes a unique instruction sequence or is not a legal byte sequence. This property holds for Y86, because every instruction has a unique combination of code and function in its initial byte, and given this byte, we can determine the length and meaning of any additional bytes. This property ensures that a processor can execute an object-code program without any ambiguity about the meaning of the code. Even if the code is embedded within other bytes in the program, we can readily determine the instruction sequence as long as we start from the first byte in the sequence. On the other hand, if we do not know the starting position of a code sequence, we cannot reliably determine how to split the sequence into individual instructions. This causes problems for disassemblers and other tools that attempt to extract machine-level programs directly from object-code byte sequences.

Practice Problem 4.1:

Determine the byte encoding of the Y86 instruction sequence that follows. The line "`.pos 0x100`" indicates that the starting address of the object code should be `0x100`.

```
.pos 0x100  # Start code at address 0x100
    irmovl $15,%ebx       #   Load 15 into %ebx
    rrmovl %ebx,%ecx      #   Copy 15 to %ecx
loop:                     # loop:
    rmmovl %ecx,-3(%ebx)  #   Save %ecx at address 15-3 = 12
    addl   %ebx,%ecx      #   Increment %ecx by 15
    jmp loop              #   Goto loop
```

Practice Problem 4.2:

For each byte sequence listed, determine the Y86 instruction sequence it encodes. If there is some invalid byte in the sequence, show the instruction sequence up to that point and indicate where the invalid value occurs. For each sequence, we show the starting address, then a colon, and then the byte sequence.

A. `0x100:30f3fcffffff40630008000000`

B. `0x200:a06f80080200000030f30a00000090`

C. `0x300:50540700000010f0b01f`

D. `0x400:6113730004000000`

E. `0x500:6362a0f0`

Aside: Comparing IA32 to Y86 instruction encodings

Compared with the instruction encodings used in IA32, the encoding of Y86 is much simpler but also less compact. The register fields occur only in fixed positions in all Y86 instructions, whereas they are packed into various positions in the different IA32 instructions. We use a 4-bit encoding of registers, even though there are only eight possible registers. IA32 uses just 3 bits. Thus, IA32 can pack a push or pop instruction into just 1 byte, with a 5-bit field indicating the instruction type and the remaining 3 bits for the register specifier. IA32 can encode constant values in 1, 2, or 4 bytes, whereas Y86 always requires 4 bytes. **End Aside.**

Aside: RISC and CISC instruction sets

IA32 is sometimes labeled as a "complex instruction set computer" (CISC—pronounced "sisk"), and is deemed to be the opposite of ISAs that are classified as "reduced instruction set computers" (RISC—pronounced "risk"). Historically, CISC machines came first, having evolved from the earliest computers. By the early 1980s, instruction sets for mainframe and minicomputers had grown quite large, as machine designers incorporated new instructions to support high-level tasks, such as manipulating circular buffers, performing decimal arithmetic, and evaluating polynomials. The first microprocessors appeared in the early 1970s and had limited instruction sets, because the integrated-circuit technology then posed severe constraints on what could be implemented on a single chip. Microprocessors evolved quickly and, by the early 1980s, were following the path of increasing instruction-set complexity set by mainframes and minicomputers. The x86 family took this path, evolving into IA32, and more recently into x86-64. Even the x86 line continues to evolve as new classes of instructions are added based on the needs of emerging applications.

The RISC design philosophy developed in the early 1980s as an alternative to these trends. A group of hardware and compiler experts at IBM, strongly influenced by the ideas of IBM researcher John Cocke, recognized that they could generate efficient code for a much simpler form of instruction set. In fact, many of the high-level instructions that were being added to instruction sets were very difficult to generate with a compiler and were seldom used. A simpler instruction set could be implemented with much less hardware and could be organized in an efficient pipeline structure, similar to those described later in this chapter. IBM did not commercialize this idea until many years later, when it developed the Power and PowerPC ISAs.

The RISC concept was further developed by Professors David Patterson, of the University of California at Berkeley, and John Hennessy, of Stanford University. Patterson gave the name RISC to this new class of machines, and CISC to the existing class, since there had previously been no need to have a special designation for a nearly universal form of instruction set.

Comparing CISC with the original RISC instruction sets, we find the following general characteristics:

CISC	Early RISC
A large number of instructions. The Intel document describing the complete set of instructions [28, 29] is over 1200 pages long.	Many fewer instructions. Typically less than 100.
Some instructions with long execution times. These include instructions that copy an entire block from one part of memory to another and others that copy multiple registers to and from memory.	No instruction with a long execution time. Some early RISC machines did not even have an integer multiply instruction, requiring compilers to implement multiplication as a sequence of additions.
Variable-length encodings. IA32 instructions can range from 1 to 15 bytes.	Fixed-length encodings. Typically all instructions are encoded as 4 bytes.
Multiple formats for specifying operands. In IA32, a memory operand specifier can have many different combinations of displacement, base and index registers, and scale factors.	Simple addressing formats. Typically just base and displacement addressing.
Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logical operations only use register operands. Memory referencing is only allowed by *load* instructions, reading from memory into a register, and *store* instructions, writing from a register to memory. This convention is referred to as a *load/store architecture*.
Implementation artifacts hidden from machine-level programs. The ISA provides a clean abstraction between programs and how they get executed.	Implementation artifacts exposed to machine-level programs. Some RISC machines prohibit particular instruction sequences and have jumps that do not take effect until the following instruction is executed. The compiler is given the task of optimizing performance within these constraints.
Condition codes. Special flags are set as a side effect of instructions and then used for conditional branch testing.	No condition codes. Instead, explicit test instructions store the test results in normal registers for use in conditional evaluation.
Stack-intensive procedure linkage. The stack is used for procedure arguments and return addresses.	Register-intensive procedure linkage. Registers are used for procedure arguments and return addresses. Some procedures can thereby avoid any memory references. Typically, the processor has many more (up to 32) registers.

The Y86 instruction set includes attributes of both CISC and RISC instruction sets. On the CISC side, it has condition codes, variable-length instructions, and stack-intensive procedure linkages. On the RISC side, it uses a load-store architecture and a regular encoding. It can be viewed as taking a CISC instruction set (IA32) and simplifying it by applying some of the principles of RISC. **End Aside.**

Value	Name	Meaning
1	`AOK`	Normal operation
2	`HLT`	`halt` instruction encountered
3	`ADR`	Invalid address encountered
4	`INS`	Invalid instruction encountered

Figure 4.5: **Y86 status codes.** In our design, the processor halts for any code other than `AOK`.

Aside: The RISC versus CISC controversy

Through the 1980s, battles raged in the computer architecture community regarding the merits of RISC versus CISC instruction sets. Proponents of RISC claimed they could get more computing power for a given amount of hardware through a combination of streamlined instruction set design, advanced compiler technology, and pipelined processor implementation. CISC proponents countered that fewer CISC instructions were required to perform a given task, and so their machines could achieve higher overall performance.

Major companies introduced RISC processor lines, including Sun Microsystems (SPARC), IBM and Motorola (PowerPC), and Digital Equipment Corporation (Alpha). A British company, Acorn Computers Ltd., developed its own architecture, ARM (originally an acronym for "Acorn RISC Machine"), which is widely used in embedded applications, such as cellphones.

In the early 1990s, the debate diminished as it became clear that neither RISC nor CISC in their purest forms were better than designs that incorporated the best ideas of both. RISC machines evolved and introduced more instructions, many of which take multiple cycles to execute. RISC machines today have hundreds of instructions in their repertoire, hardly fitting the name "reduced instruction set machine." The idea of exposing implementation artifacts to machine-level programs proved to be short-sighted. As new processor models were developed using more advanced hardware structures, many of these artifacts became irrelevant, but they still remained part of the instruction set. Still, the core of RISC design is an instruction set that is well-suited to execution on a pipelined machine.

More recent CISC machines also take advantage of high-performance pipeline structures. As we will discuss in Section 5.7, they fetch the CISC instructions and dynamically translate them into a sequence of simpler, RISC-like operations. For example, an instruction that adds a register to memory is translated into three operations: one to read the original memory value, one to perform the addition, and a third to write the sum to memory. Since the dynamic translation can generally be performed well in advance of the actual instruction execution, the processor can sustain a very high execution rate.

Marketing issues, apart from technological ones, have also played a major role in determining the success of different instruction sets. By maintaining compatibility with its existing processors, Intel with x86 made it easy to keep moving from one generation of processor to the next. As integrated-circuit technology improved, Intel and other x86 processor manufacturers could overcome the inefficiencies created by the original 8086 instruction set design, using RISC techniques to produce performance comparable to the best RISC machines. As we saw in Section 3.13, the evolution of IA32 into x86-64 provided an opportunity to incorporate several features of RISC into x86. In the areas of desktop and laptop computing, x86 has achieved total domination, and it is increasingly popular for high-end server machines.

RISC processors have done very well in the market for *embedded processors*, controlling such systems as cellular telephones, automobile brakes, and Internet appliances. In these applications, saving on cost and power is more important than maintaining backward compatibility. In terms of the number of processors sold, this is a very large and growing market. **End Aside.**

IA32 code

```
    int Sum(int *Start, int Count)
Sum:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%ecx     ecx = Start
  movl 12(%ebp),%edx    edx = Count
  xorl %eax,%eax        sum = 0
  testl %edx,%edx
  je .L34
.L35:
  addl (%ecx),%eax    add *Start to sum

  addl $4,%ecx        Start++

  decl %edx           Count--

  jnz .L35            Stop when 0
.L34:
  movl %ebp,%esp
  popl %ebp
  ret
```

Y86 code

```
    int Sum(int *Start, int Count)
Sum:
  pushl %ebp
  rrmovl %esp,%ebp
  mrmovl 8(%ebp),%ecx       ecx = Start
  mrmovl 12(%ebp),%edx      edx = Count
  xorl %eax,%eax            sum = 0
  andl   %edx,%edx          Set condition codes
  je     End
Loop:
  mrmovl (%ecx),%esi        get *Start
  addl %esi,%eax            add to sum
  irmovl $4,%ebx
  addl %ebx,%ecx            Start++
  irmovl $-1,%ebx
  addl %ebx,%edx            Count--
  jne    Loop               Stop when 0
End:
  rrmovl %ebp,%esp
  popl %ebp
  ret
```

Figure 4.6: **Comparison of Y86 and IA32 assembly programs.** The Sum function computes the sum of an integer array. The Y86 code differs from the IA32 mainly in that it may require multiple instructions to perform what can be done with a single IA32 instruction.

4.1.4 Y86 Exceptions

The programmer-visible state for Y86 (Figure 4.1) includes a status code Stat describing the overall state of the executing program. The possible values for this code are shown in Figure 4.5. Code value 1, named AOK, indicates that the program is executing normally, while the other codes indicate that some type of *exception* has occurred. Code 2, named HLT, indicates that the processor has executed a halt instruction. Code 3, named ADR, indicates that the processor attempted to read from or write to an invalid memory address, either while fetching an instruction or while reading or writing data. We limit the maximum address (the exact limit varies by implementation), and any access to an address beyond this limit will trigger an ADR exception. Code 4, named INS, indicates that an invalid instruction code has been encountered.

For Y86, we will simply have the processor stop executing instructions when it encounters any of the exceptions listed. In a more complete design, the processor would typically invoke an *exception handler*, a procedure designated to handle the specific type of exception encountered. As described in Chapter 8, exception handlers can be configured to have different effects, such as aborting the program or invoking a user-defined *signal handler*.

4.1.5 Y86 Programs

Figure 4.6 shows IA32 and Y86 assembly code for the following C function:

```
int Sum(int *Start, int Count)
{
    int sum = 0;
    while (Count) {
        sum += *Start;
        Start++;
        Count--;
    }
    return sum;
}
```

The IA32 code was generated by the GCC compiler. The Y86 code is essentially the same, except that Y86 sometimes requires two instructions to accomplish what can be done with a single IA32 instruction. If we had written the program using array indexing, however, the conversion to Y86 code would be more difficult, since Y86 does not have scaled addressing modes. This code follows many of the programming conventions we have seen for IA32, including the use of the stack and frame pointers. For simplicity, it does not follow the IA32 convention of having some registers designated as callee-save registers. This is just a programming convention that we can either adopt or ignore as we please.

Figure 4.7 shows an example of a complete program file written in Y86 assembly code. The program contains both data and instructions. Directives indicate where to place code or data and how to align it. The program specifies issues such as stack placement, data initialization, program initialization, and program termination.

In this program, words beginning with "`.`" are *assembler directives* telling the assembler to adjust the address at which it is generating code or to insert some words of data. The directive `.pos 0` (line 2) indicates that the assembler should begin generating code starting at address `0`. This is the starting address for all Y86 programs. The next two instructions (lines 3 and 4) initialize the stack and frame pointers. We can see that the label `Stack` is declared at the end of the program (line 47), to indicate address `0x100` using a `.pos` directive (line 46). Our stack will therefore start at this address and grow toward lower addresses. We must ensure that the stack does not grow so large that it overwrites the code or other program data.

Lines 9 to 13 of the program declare an array of four words, having values `0xd`, `0xc0`, `0xb00`, and `0xa000`. The label `array` denotes the start of this array, and is aligned on a 4-byte boundary (using the `.align` directive). Lines 15 to 24 show a "main" procedure that calls the function `Sum` on the 4-word array and then halts.

As this example shows, since our only tool for creating Y86 code is an assembler, the programmer must perform tasks we ordinarily delegate to the compiler, linker, and run-time system. Fortunately, we only do this for small programs, for which simple mechanisms suffice.

Figure 4.8 shows the result of assembling the code shown in Figure 4.7 by an assembler we call YAS. The assembler output is in ASCII format to make it more readable. On lines of the assembly file that contain instructions or data, the object code contains an address, followed by the values of between 1 and 6 bytes.

```
# Execution begins at address 0
        .pos 0
init:   irmovl Stack, %esp      # Set up stack pointer
        irmovl Stack, %ebp      # Set up base pointer
        call Main               # Execute main program
        halt                    # Terminate program

# Array of 4 elements
        .align 4
array:  .long 0xd
        .long 0xc0
        .long 0xb00
        .long 0xa000

Main:   pushl %ebp
        rrmovl %esp,%ebp
        irmovl $4,%eax
        pushl %eax              # Push 4
        irmovl array,%edx
        pushl %edx              # Push array
        call Sum                # Sum(array, 4)
        rrmovl %ebp,%esp
        popl %ebp
        ret

        # int Sum(int *Start, int Count)
Sum:    pushl %ebp
        rrmovl %esp,%ebp
        mrmovl 8(%ebp),%ecx     # ecx = Start
        mrmovl 12(%ebp),%edx    # edx = Count
        xorl %eax,%eax          # sum = 0
        andl   %edx,%edx        # Set condition codes
        je     End
Loop:   mrmovl (%ecx),%esi      # get *Start
        addl %esi,%eax          # add to sum
        irmovl $4,%ebx          #
        addl %ebx,%ecx          # Start++
        irmovl $-1,%ebx         #
        addl %ebx,%edx          # Count--
        jne    Loop             # Stop when 0
End:    rrmovl %ebp,%esp
        popl %ebp
        ret

# The stack starts here and grows to lower addresses
        .pos 0x100
Stack:
```

Figure 4.7: **Sample program written in Y86 assembly code.** The `Sum` function is called to compute the sum of a four-element array.

```
                    | # Execution begins at address 0
  0x000:            |         .pos 0
  0x000: 30f400010000 | init:   irmovl Stack, %esp      # Set up stack pointer
  0x006: 30f500010000 |         irmovl Stack, %ebp      # Set up base pointer
  0x00c: 8024000000   |         call Main               # Execute main program
  0x011: 00           |         halt                    # Terminate program
                      |
                      | # Array of 4 elements
  0x014:              |         .align 4
  0x014: 0d000000     | array:  .long 0xd
  0x018: c0000000     |         .long 0xc0
  0x01c: 000b0000     |         .long 0xb00
  0x020: 00a00000     |         .long 0xa000
                      |
  0x024: a05f         | Main:   pushl %ebp
  0x026: 2045         |         rrmovl %esp,%ebp
  0x028: 30f004000000 |         irmovl $4,%eax
  0x02e: a00f         |         pushl %eax              # Push 4
  0x030: 30f214000000 |         irmovl array,%edx
  0x036: a02f         |         pushl %edx              # Push array
  0x038: 8042000000   |         call Sum                # Sum(array, 4)
  0x03d: 2054         |         rrmovl %ebp,%esp
  0x03f: b05f         |         popl %ebp
  0x041: 90           |         ret
                      |
                      |         # int Sum(int *Start, int Count)
  0x042: a05f         | Sum:    pushl %ebp
  0x044: 2045         |         rrmovl %esp,%ebp
  0x046: 501508000000 |         mrmovl 8(%ebp),%ecx     # ecx = Start
  0x04c: 50250c000000 |         mrmovl 12(%ebp),%edx    # edx = Count
  0x052: 6300         |         xorl %eax,%eax          # sum = 0
  0x054: 6222         |         andl   %edx,%edx        # Set condition codes
  0x056: 7378000000   |         je     End
  0x05b: 506100000000 | Loop:   mrmovl (%ecx),%esi      # get *Start
  0x061: 6060         |         addl %esi,%eax          # add to sum
  0x063: 30f304000000 |         irmovl $4,%ebx          #
  0x069: 6031         |         addl %ebx,%ecx          # Start++
  0x06b: 30f3ffffffff |         irmovl $-1,%ebx         #
  0x071: 6032         |         addl %ebx,%edx          # Count--
  0x073: 745b000000   |         jne    Loop             # Stop when 0
  0x078: 2054         | End:    rrmovl %ebp,%esp
  0x07a: b05f         |         popl %ebp
  0x07c: 90           |         ret
                      |
                      | # The stack starts here and grows to lower addresses
  0x100:              |         .pos 0x100
  0x100:              | Stack:
```

Figure 4.8: **Output of YAS assembler.** Each line includes a hexadecimal address and between 1 and 6 bytes of object code.

We have implemented an *instruction set simulator* we call YIS, the purpose of which is to model the execution of a Y86 machine-code program, without attempting to model the behavior of any specific processor implementation. This form of simulation is useful for debugging programs before actual hardware is available, and for checking the result of either simulating the hardware or running the program on the hardware itself. Running on our sample object code, YIS generates the following output:

```
Stopped in 52 steps at PC = 0x11.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x0000abcd
%ecx:   0x00000000      0x00000024
%ebx:   0x00000000      0xffffffff
%esp:   0x00000000      0x00000100
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x0000a000

Changes to memory:
0x00e8: 0x00000000      0x000000f8
0x00ec: 0x00000000      0x0000003d
0x00f0: 0x00000000      0x00000014
0x00f4: 0x00000000      0x00000004
0x00f8: 0x00000000      0x00000100
0x00fc: 0x00000000      0x00000011
```

The first line of the simulation output summarizes the execution and the resulting values of the PC and program status. In printing register and memory values, it only prints out words that change during simulation, either in registers or in memory. The original values (here they are all zero) are shown on the left, and the final values are shown on the right. We can see in this output that register `%eax` contains `0xabcd`, the sum of the 4-element array passed to subroutine `Sum`. In addition, we can see that the stack, which starts at address `0x100` and grows toward lower addresses, has been used, causing changes to words of memory at addresses `0xe8` through `0xfc`. This is well away from `0x7c`, the maximum address of the executable code.

Practice Problem 4.3:

Write Y86 code to implement a recursive sum function `rSum`, based on the following C code:

```
int rSum(int *Start, int Count)
{
    if (Count <= 0)
        return 0;
    return *Start + rSum(Start+1, Count-1);
}
```

You might find it helpful to compile the C code on an IA32 machine and then translate the instructions to Y86.