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 one 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/Motorola PowerPC, and Sun Microsystems SPARC 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.* 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 as simple a structure as possible 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 containing 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 800 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

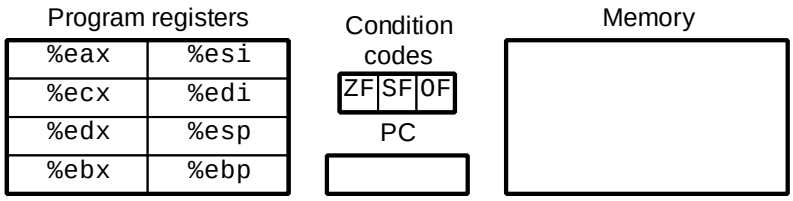


Figure 4.1: **Y86 programmer-visible state.** As with IA32, programs for Y86 access and modify the program registers, the condition code, the program counter (PC), and the memory.

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 editting these files and recompiling the simulator, you can alter and extend the simulation 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.

4.1 The Y86 Instruction Set Architecture

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 10. For now, we can think of the virtual memory system as providing Y86 programs with an image of a monolithic byte array.

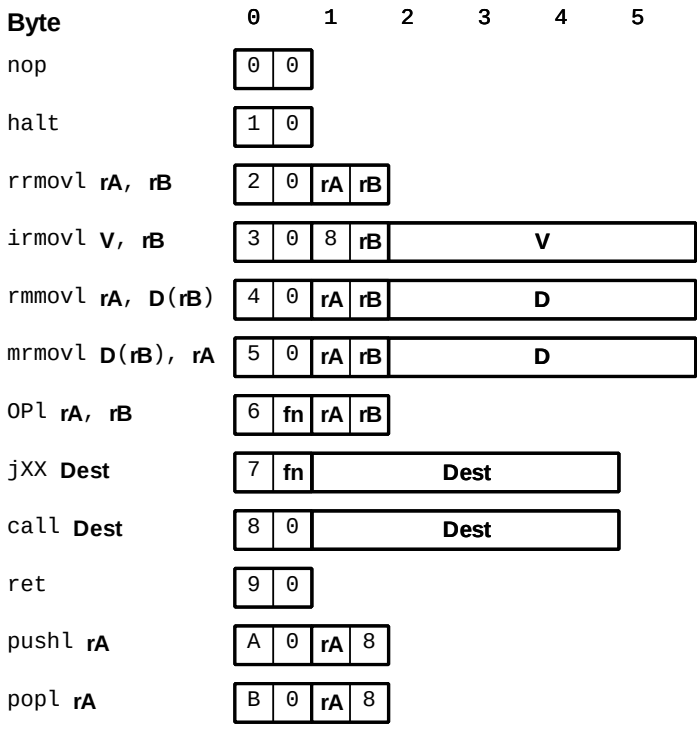


Figure 4.2: **Y86 instruction set.** Instruction encodings range between 1 and 6 bytes. An instruction consists of a one-byte instruction specifier, possibly a one-byte register specifier, and possibly a four-byte constant word. Field `fn` specifies a particular integer operation (`OPl`) or a particular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

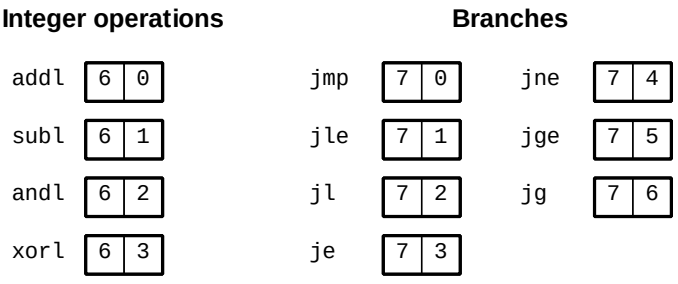


Figure 4.3: **Function codes for Y86 instruction set.** The codes specify a particular integer operation or branch condition. These instructions are shown as `OPl` and `jXX` in Figure 4.2.

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 four-byte integer operations; it has fewer addressing modes; and it includes a smaller set of operations. Since we only use four-byte data, we refer to these as "words." 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 is similar to the GAS representation of IA32 programs.

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 the register value in the address computation.

  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 3.11).

- 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 stop. We use `halt` in our Y86 programs to stop the simulator.

Figure 4.2 also shows the byte-level encoding of the instructions. Each instruction requires between one and six bytes, depending on which fields are required. Every instruction has an initial byte identifying the instruction type. This byte is split into two four-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 hexadecimal `B`. 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 and branch instructions.

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

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

Figure 4.4: **Y86 program register identifiers.** Each of the eight program registers has an associated identifer (ID) ranging from 0 to 7. ID 8 in a register field of an instruction indicates the absence of a register operand.

stored within the CPU in a *register file*, a small random-access memory where the register IDs serve as addresses. ID value 8 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 one 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, 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 8. This convention will prove useful in our processor implementation.

Some instructions require an additional four-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 four-byte constant word. We first pad `0x12345` with leading 0s to fill out four 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 generating code at address 0x100
    irmovl $15,%ebx
    rrmovl %ebx,%ecx
loop:
    rmmovl %ecx,-3(%ebx)
    addl   %ebx,%ecx
    jmp 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. `0x100:3083fcffffff40630008000010`
B. `0x200:a06880080200001030830a00000090`
C. `0x300:50540700000000f0b018`
D. `0x400:6113730004000010`
E. `0x500:6362a080`

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 only occur in fixed positions in all Y86 instructions, whereas they are packed into various positions in the different IA32 instructions. We use a four-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 one 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, peforming 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 80x86 family took this path, evolving into IA32. Even IA32 continues to evolve as new classes of instructions are added to support the processing required by multimedia 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 [19] is over 700 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 four bytes.
Multiple formats for specifying operands. In IA32, a memory operand specifi er can have many different combinations of displacement, base and index registers, and scale factors.	Simple addressing formats. Typically just base and displacment addressing.
Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logial 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 that store the test result in a normal register are used for 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 addressess. 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.**

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 instruction 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).

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 fi tting the name "reduced instruction set machine." The idea of exposing implementation

IA32 code

```
     int Sum(int *Start, int Count)
 1 Sum:
 2   pushl %ebp
 3   movl %esp,%ebp
 4   movl 8(%ebp),%ecx    ecx = Start
 5   movl 12(%ebp),%edx   edx = Count
 6   xorl %eax,%eax       sum = 0
 7   testl %edx,%edx
 8   je .L34
 9 .L35:
10   addl (%ecx),%eax     add *Start to sum
11   addl $4,%ecx         Start++
12   decl %edx            Count--
13   jnz .L35             Stop when 0
14 .L34:
15   movl %ebp,%esp
16   popl %ebp
17   ret
```

Y86 code

```
     int Sum(int *Start, int Count)
 1 Sum:    pushl %ebp
 2   rrmovl %esp,%ebp
 3   mrmovl 8(%ebp),%ecx       ecx = Start
 4   mrmovl 12(%ebp),%edx      edx = Count
 5   irmovl $0, %eax           sum = 0
 6   andl   %edx,%edx
 7   je     End
 8 Loop:   mr-
movl (%ecx),%esi        get *Start
 9   addl %esi,%eax            add to sum
10   irmovl $4,%ebx
11   addl %ebx,%ecx            Start++
12   irmovl $-1,%ebx
13   addl %ebx,%edx            Count-
                               -
14   jne    Loop               Stop when 0
15 End:
16   popl %ebp
17   ret
```

Figure 4.5: **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.

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 IA32 made it easy to keep moving from one generation of processor to the next. As integrated circuit technology improved, Intel and other IA32 processor manufacturers could overcome the ineffi ciencies created by the original 8086 instruction-set design, using RISC techniques to produce performance comparable to the best RISC machines. In the areas of desktop and laptop computing, IA32 has achieved total domination.

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.**

Figure 4.5 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 C compiler GCC. 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.

Figure 4.6 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 point of 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 39), to indicate address `0x100` using a `.pos` directive (line 38). Our stack will therefore start at this address and grow downward.

Lines 8 to 12 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 four-byte boundary (using the `.align` directive). Lines 14 to 19 show a "main" procedure that calls the function `Sum` on the four-word array and then halts.

As this example shows, writing a program in Y86 requires the programmer to perform tasks we ordinarily assign to the compiler, linker, and run-time system. Fortunately, we only do this for small programs for which simple mechanisms suffice.

Figure 4.7 shows the result of assembling the code shown in Figure 4.6 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.

We have implemented an instruction set simulator we call YIS. Running on our sample object code, it generates the following output:

```
Stopped in 46 steps at PC = 0x3a.  Exception 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x0000abcd
%ecx:   0x00000000      0x00000024
%ebx:   0x00000000      0xffffffff
%esp:   0x00000000      0x000000f8
```

code/arch/y86-code/asum.ys

```
1  # Execution begins at address 0
2          .pos 0
3  init:   irmovl Stack, %esp      # Set up Stack pointer
4          irmovl Stack, %ebp      # Set up base pointer
5          jmp Main                # Execute main program
6
7  # Array of 4 elements
8          .align 4
9  array:  .long 0xd
10         .long 0xc0
11         .long 0xb00
12         .long 0xa000
13
14 Main:   irmovl $4,%eax
15         pushl %eax      # Push 4
16         irmovl array,%edx
17         pushl %edx      # Push array
18         call Sum        # Sum(array, 4)
19         halt
20
21         # int Sum(int *Start, int Count)
22 Sum:    pushl %ebp
23         rrmovl %esp,%ebp
24         mrmovl 8(%ebp),%ecx     # ecx = Start
25         mrmovl 12(%ebp),%edx    # edx = Count
26         irmovl $0, %eax         # sum = 0
27         andl   %edx,%edx
28         je     End
29 Loop:   mrmovl (%ecx),%esi      # get *Start
30         addl %esi,%eax          # add to sum
31         irmovl $4,%ebx          #
32         addl %ebx,%ecx          # Start++
33         irmovl $-1,%ebx         #
34         addl %ebx,%edx          # Count--
35         jne    Loop             # Stop when 0
36 End:
37         popl %ebp
38         ret
39         .pos 0x100
40 Stack:  # The stack goes here
```

code/arch/y86-code/asum.ys

Figure 4.6: **Sample program written in Y86 assembly code.** The Sum function is called to compute the sum of a 4-element array.

code/arch/y86-code/asum.yo

```
                    | # Execution begins at address 0
  0x000:            |         .pos 0
  0x000: 308400010000 | init:   irmovl Stack, %esp      # Set up Stack pointer
  0x006: 308500010000 |         irmovl Stack, %ebp      # Set up base pointer
  0x00c: 7024000000   |         jmp Main                # Execute main 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: 308004000000 | Main:   irmovl $4,%eax
  0x02a: a008         |         pushl %eax      # Push 4
  0x02c: 308214000000 |         irmovl array,%edx
  0x032: a028         |         pushl %edx      # Push array
  0x034: 803a000000   |         call Sum        # Sum(array, 4)
  0x039: 10           |         halt
                    |
                    |         # int Sum(int *Start, int Count)
  0x03a: a058         | Sum:    pushl %ebp
  0x03c: 2045         |         rrmovl %esp,%ebp
  0x03e: 501508000000 |         mrmovl 8(%ebp),%ecx     # ecx = Start
  0x044: 50250c000000 |         mrmovl 12(%ebp),%edx    # edx = Count
  0x04a: 308000000000 |         irmovl $0, %eax         # sum = 0
  0x050: 6222         |         andl   %edx,%edx
  0x052: 7374000000   |         je     End
  0x057: 506100000000 | Loop:   mrmovl (%ecx),%esi      # get *Start
  0x05d: 6060         |         addl %esi,%eax          # add to sum
  0x05f: 308304000000 |         irmovl $4,%ebx          #
  0x065: 6031         |         addl %ebx,%ecx          # Start++
  0x067: 3083ffffffff |         irmovl $-1,%ebx         #
  0x06d: 6032         |         addl %ebx,%edx          # Count--
  0x06f: 7457000000   |         jne    Loop             # Stop when 0
  0x074:            | End:
  0x074: b058         |         popl %ebp
  0x076: 90           |         ret
  0x100:            |         .pos 0x100
  0x100:            | Stack:  # The stack goes here
```

code/arch/y86-code/asum.yo

Figure 4.7: **Output of YAS assembler.** Each line includes a hexadecimal address and between 1 and 6 bytes of object code.

```
%ebp:   0x00000000      0x00000100
%esi:   0x00000000      0x0000a000

Changes to memory:
0x00f0: 0x00000000      0x00000100
0x00f4: 0x00000000      0x00000039
0x00f8: 0x00000000      0x00000014
0x00fc: 0x00000000      0x00000004
```

The simulator only prints out words that change during simulation, either in registers or in memory. The original values (here they are all 0) 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 four-element array passed to subroutine `Sum`. In addition, we can see that the stack, which starts at address `0x100` and grows downward, has been used, causing changes to memory at addresses `0xf0` through `0xfc`.

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.

Practice Problem 4.4:

The `pushl` instruction both decrements the stack pointer by 4 and writes a register value to memory. It is not totally clear what the processor should do with the instruction `pushl %esp`, since the register being pushed is being changed by the same instruction. Two conventions are possible: (1) push the original value of `%esp`, or (2) push the decremented value of `%esp`.

Let's resolve this issue by doing the same thing an IA32 processor would do. We could try reading the Intel documentation on this instruction, but a simpler approach is to conduct an experiment on an actual machine. The C compiler would not normally generate this instruction, so we must use hand-generated assembly code for this task. As described in Section 3.15, the best way to insert small amounts of assembly code into a C program is to use the `asm` feature of GCC. Here is a test program we have written. Rather than attempting to read the `asm` declaration, you will find it easiest to read the assembly code in the comment preceding it.

```
int pushtest()
{
    int rval;
    /* Insert the following assembly code:
       movl %esp,%eax   # Save stack pointer
```

```
       pushl %esp       # Push stack pointer
       popl  %edx       # Pop it back
       subl  %edx,%eax  # 0 or 4
       movl  %eax,rval  # Set as return value
    */
  asm("movl %%esp,%%eax;pushl %%esp;popl %%edx;subl %%edx,%%eax;movl %%eax,%0"
        : "=r" (rval)
        : /* No Input */
        : "%edx", "%eax");
    return rval;
}
```

In our experiments, we find that the function `pushtest` returns 0. What does this imply about the behavior of the instruction `pushl %esp` under IA32?

Practice Problem 4.5:

A similar ambiguity occurs for the instruction `popl %esp`. It could either set `%esp` to the value read from memory or to the incremented stack pointer. As with Practice Problem 4.4, let us run an experiment to determine how an IA32 machine would handle this instruction and then design our Y86 machine to follow the same convention.

```
int poptest(int tval)
{
    int rval;
    /* Insert the following assembly code:
       pushl tval     # Save tval on stack
       movl %esp,%edx # Save stack pointer
       popl %esp      # Pop to stack pointer.
       movl %esp,rval # Set popped value as return value
       movl %edx,%esp # Restore original stack pointer
    */
    asm("pushl %1; movl %%esp,%%edx; popl %%esp; movl %%esp,%0; movl %%edx,%%esp"
        : "=r" (rval)
        : "r" (tval)
        : "%edx");
    return rval;
}
```

We find this function always returns `tval`, the value passed to it as its argument. What does this imply about the behavior of `popl %esp`? What other Y86 instruction would have the exact same behavior?

4.2 Logic Design and the Hardware Control Language HCL

In hardware design, electronic circuits are used to compute functions on bits and to store bits in different kinds of memory elements. Most contemporary circuit technology represents different bit values as high or

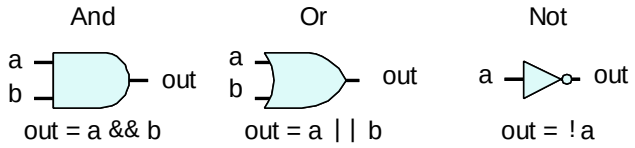


Figure 4.8: **Logic gate types.** Each gate generates output equal to some Boolean function of its inputs.

low voltages on signal wires. In current technology, logic value 1 is represented by a high voltage of around 1.0 volt, while logic value 0 is represented by a low voltage of around 0.0 volts. Three major components are required to implement a digital system: combinational logic to compute functions on the bits, memory elements to store bits, and clock signals to regulate the updating of the memory elements.

In this section, we provide a brief description of these different components. We also introduce HCL (for "hardware control language"), the language that we use to describe the control logic of the different processor designs. We only describe HCL informally here. A complete reference for HCL can be found in Appendix A.

Aside: Modern Logic Design
At one time hardware designers created circuit designs by drawing schematic diagrams of logic circuits (first with paper and pencil and later with computer graphics terminals). Nowadays, most designs are expressed in a *hardware description language* (HDL), a textual notation that looks similar to a programming language but that is used to describe hardware structures rather than program behaviors. The most commonly used languages are Verilog, having a syntax similar to C, and VHDL, having a syntax similar to the Ada programming language. These languages were originally designed for expressing simulation models of digital circuits. In the mid-1980s, researchers developed *logic synthesis* programs that could generate efficient circuit designs from HDL descriptions. There are now a number of commercial synthesis programs, and this has become the dominant technique for generating digital circuits. This shift from hand-designed circuits to synthesized ones can be likened to the shift from writing programs in assembly code to writing them in a high-level language and having a compiler generate the machine code. **End Aside.**

4.2.1 Logic Gates

Logic gates are the basic computing elements for digital circuits. They generate an output equal to some Boolean function of the bit values at their inputs. Figure 4.8 shows the standard symbols used for Boolean functions AND, OR, and NOT. HCL expressions are shown below the gates for the Boolean operations. As you can see, we adopt the syntax for logic operators in C (Section 2.1.9): && for AND, || for OR, and ! for NOT. We use these instead of the bit-level C operators &, |, and ~, because logic gates operate on single-bit quantities, not entire words.

Logic gates are always active. If some input to a gate changes, then within some small amount of time, the output will change accordingly.

4.2.2 Combinational Circuits and HCL Boolean Expressions

By assembling a number of logic gates into a network, we can construct computational blocks known as *combinational circuits*. Two restrictions are placed on how the networks are constructed:

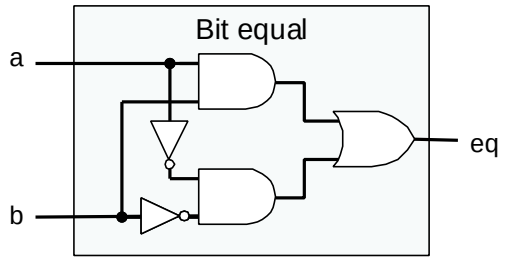


Figure 4.9: **Combinational circuit to test for bit equality.** The output will equal 1 when both inputs are 0, or both are 1.

- The outputs of two or more logic gates cannot be connected together. Otherwise the two could try to drive the wire in opposite directions, possibly causing an invalid voltage or a circuit malfunction.
- The network must be *acyclic*. That is, there cannot be a path through a series of gates that forms a loop in the network. Such loops can cause ambiguity in the function computed by the network.

Figure 4.9 shows an example of a simple combinational circuit that we will find useful. It has two inputs, a and b. It generates a single output eq, such that the output will equal 1 if either a and b are both 1 (detected by the upper AND gate) or are both 0 (detected by the lower AND gate). We write the function of this network in HCL as:

```
bool eq = (a && b) || (!a && !b);
```

This code simply defines the bit-level (denoted by data type `bool`) signal eq as a function of inputs a and b. As this example shows HCL uses C-style syntax, with '=' associating a signal name with an expression. Unlike C, however, we do not view this as performing a computation and assigning the result to some memory location. Instead, it is simply a way to give a name to an expression.

Practice Problem 4.6:

Write an HCL expression for a signal xor, equal to the EXCLUSIVE-OR of inputs a and b. What is the relation between the signals xor and eq defined above?

Figure 4.10 shows another example of a simple but useful combinational circuit known as a *multiplexor*. A multiplexor selects a value from among a set of different data signals, depending on the value of a control input signal. In this single-bit multiplexor, the two data signals are the input bits a and b, while the control signal is the input bit s. The output will equal a when s is 1, and it will equal b when s is 0. In this circuit, we can see that the two AND gates determine whether to pass their respective data inputs to the OR gate. The upper AND gate passes signal b when s is 0 (since the other input to the gate is !s), while the lower AND gate passes signal a when s is 1. Again, we can write an HCL expression for the output signal, using the same operations as are present in the combinational circuit:

```
bool out = (s && a) || (!s && b);
```

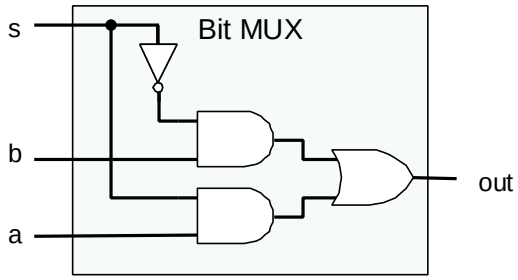


Figure 4.10: **Single-bit multiplexor circuit.** The output will equal input a if the control signal s is 1 and will equal input b when s is 0.

Our HCL expressions demonstrate a clear parallel between combinational logic circuits and logical expressions in C. They both use Boolean operations to compute functions over their inputs. Several differences between these two ways of expressing computation are worth noting:

- Since a combinational circuit consists of a series of logic gates, it has the property that the outputs continually respond to changes in the inputs. If some input to the circuit changes, then after some delay, the outputs will change accordingly. In contrast, a C expression is only evaluated when it is encountered during the execution of a program.
- Logical expressions in C allow arguments to be arbitrary integers, interpreting 0 as FALSE and anything else as TRUE. In contrast, our logic gates only operate over the bit values 0 and 1.
- Logical expressions in C have the property that they might only be partially evaluated. If the outcome of an AND or OR operation can be determined by just evaluating the first argument, then the second argument will not be evaluated. For example, with the C expression:

  ```
  (a && !a) && func(b,c)
  ```

  the function `func` will not be called, because the expression `(a && !a)` evaluates to 0. In contrast, combinational logic does not have any partial evaluation rules. The gates simply respond to changes on their inputs.

4.2.3 Word-Level Combinational Circuits and HCL Integer Expressions

By assembling large networks of logic gates, we can construct combinational circuits that compute much more complex functions. Typically, we design circuits that operate on data *words*. These are groups of bit-level signals that represent an integer or some control pattern. For example, our processor designs will contain numerous words, with word sizes ranging between 4 and 32 bits, representing integers, addresses, instruction codes, and register identifiers.

Combinational circuits to perform word-level computations are constructed using logic gates to compute the individual bits of the output word, based on the individual bits of the input word. For example, Figure 4.11 shows a combinational circuit that tests whether two 32-bit words A and B are equal. That is, the output will

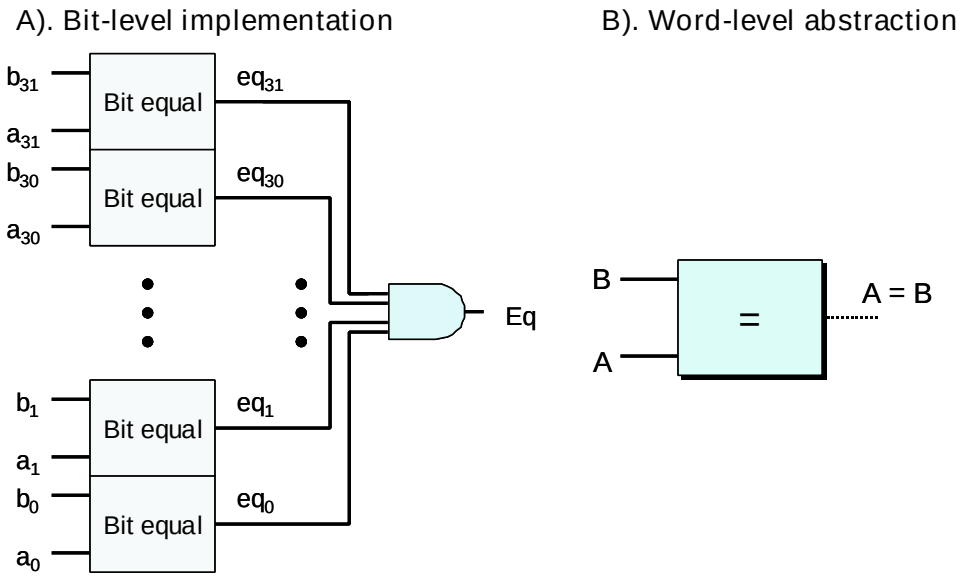


Figure 4.11: **Word-level equality test circuit.** The output will equal 1 when each bit from word A equals its counterpart from word B. Word-level equality is one of the operations in HCL.

equal 1 if and only if each bit of A equals the corresponding bit of B. This circuit is implemented using 32 of the single-bit equality circuits shown in Figure 4.9. The outputs of these single-bit circuits are combined with an AND gate to form the circuit output.

In HCL, we will declare any word-level signal as an `int`, without specifying the word size. This is done for simplicity. In a full-featured hardware description language, every word can be declared to have a specific number of bits. HCL allows words to be compared for equality, and so the functionality of the circuit shown in Figure 4.11 can be expressed at the word level as

```
bool Eq = (A == B);
```

where arguments A and B are of type `int`. Note that we use the same syntax conventions as in C, where '=' denotes assignment, while '==' denotes the equality operator.

As is shown on the right side of Figure 4.11, we will draw word-level circuits using medium-thickness lines to represent the set of wires carrying the individual bits of the word, and we will show the resulting Boolean signal as a dashed line.

Practice Problem 4.7:

Suppose you want to implement a word-level equality circuit using the EXCLUSIVE-OR circuits from Practice Problem 4.6 rather than from bit-level equality circuits. Design such a circuit for a 32-bit word consisting of 32 bit-level EXCLUSIVE-OR circuits and two additional logic gates.

Figure 4.12 shows the circuit for a word-level multiplexor. This circuit generates a 32-bit word Out equal to one of the two input words, A or B, depending on the control input bit s. The circuit consists of 32

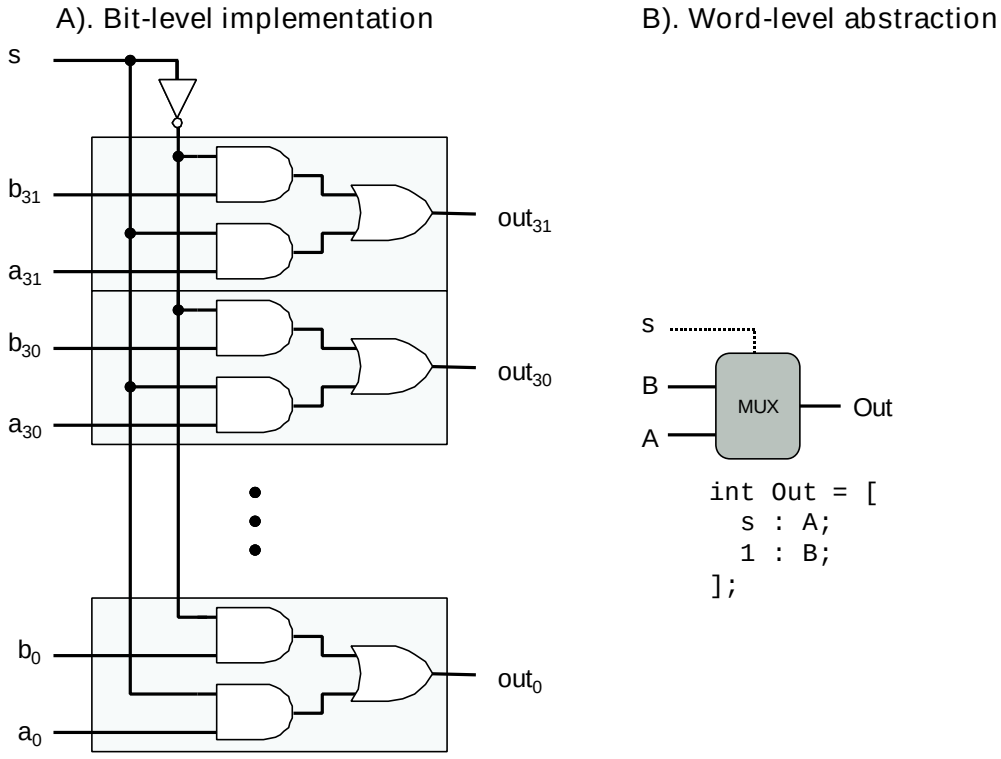


Figure 4.12: **Word-level multiplexor circuit.** The output will equal input word A when the control signal s is 1, and it will equal B otherwise. Multiplexors are described in HCL using case expressions.

identical subcircuits, each having a structure similar to the bit-level multiplexor from Figure 4.10. Rather than simply replicating the bit-level multiplexor 32 times, the word-level version reduces the number of inverters by generating `!s` once and reusing it at each bit position.

We will use many forms of multiplexors in our processor designs. They allow us to select a word from a number of sources depending on some control condition. Multiplexing functions are described in HCL using *case expressions*. A case expression has the following general form:

$$
\begin{array}{l}
\texttt{[} \\
\quad \begin{array}{lcl} select_1 & : & expr_1 \\ select_2 & : & expr_2 \\ & \vdots & \\ select_k & : & expr_k \end{array} \\
\texttt{]}
\end{array}
$$

The expression contains a series of cases, where each case i consists of a Boolean expression $select_i$, indicating when this case should be selected, and an integer expression $expr_i$, indicating the resulting value.

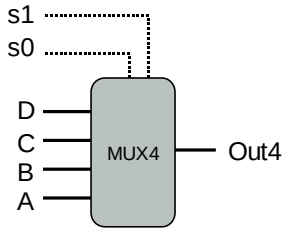
Unlike the switch statement of C, we do not require the different selection expressions to be mutually exclusive. Logically, the selection expressions are evaluated in sequence, and the case for the first one yielding 1 is selected. For example, the word-level multiplexor of Figure 4.12 can be described in HCL as:

```
int Out = [
        s: A;
        1: B;
];
```

In this code, the second selection expression is simply `1`, indicating that this case should be selected if no prior one has been. This is the way to specify a default case in HCL. Nearly all case expressions end in this manner.

Allowing nonexclusive selection expressions makes the HCL code more readable. An actual hardware multiplexor must have mutually exclusive signals controlling which input word should be passed to the output, such as the signals `s` and `!s` in Figure 4.12. To translate an HCL case expression into hardware, a logic synthesis program would need to analyze the set of selection expressions and resolve any possible conflicts by making sure that only the first matching case would be selected.

The selection expressions can be arbitrary Boolean expressions, and there can be an arbitrary number of cases. This allows case expressions to describe blocks where there are many choices of input signals with complex selection criteria. For example, consider the following diagram of a four-way multiplexor:

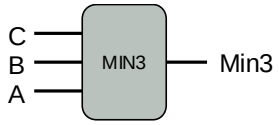


This circuit selects from among the four input words A, B, C, and D based on the control signals s1 and s0, treating the controls as a two-bit binary number. We can express this in HCL using Boolean expressions to describe the different combinations of control bit patterns:

```
int Out4 = [
        !s1 && !s0 : A; # 00
        !s1        : B; # 01
        s1 && !s0  : C; # 10
        1          : D; # 11
];
```

The comments on the right (any text starting with # and running for the rest of the line is a comment) show which combination of s1 and s0 will cause the case to be selected. Observe that the selection expressions can sometimes be simplified, since only the first matching case is selected. For example, the second expression can be written !s1, rather than the more complete !s1 && s0, since the only other possibility having s1 equal to 0 was given as the first selection expression.

As a final example, suppose we want design a logic circuit that finds the minimum value among a set of words A, B, and C, diagrammed as follows:



We can express this using an HCL case expression as

```
int Min3 = [
        A <= B && A <= C : A;
        B <= A && B <= C : B;
        1                : C;
];
```

Practice Problem 4.8:

Write HCL code describing a circuit that for word inputs A, B, and C selects the *median* of the three values. That is, the output equals the word lying between the minimum and maximum of the three inputs.

Combinational logic circuits can be designed to perform many different types of operations on word-level data. The detailed design of these is beyond the scope of our presentation. One important combinational circuit, known as an *arithmetic/logic unit* (ALU), is diagrammed at an abstract level in Figure 4.13. This circuit has three inputs: two data inputs labeled A and B, and a control input. Depending on the setting of the control input, the circuit will perform different arithmetic or logical operations on the data inputs. Observe that the four operations diagrammed for this ALU correspond to the four different integer operations supported by the Y86 instruction set, and the control values match the function codes for these instructions (Figure 4.3). Note also the ordering of operands for subtraction, where the A input is subtracted from the B input. This ordering is chosen in anticipation of the ordering of arguments in the subl instruction.

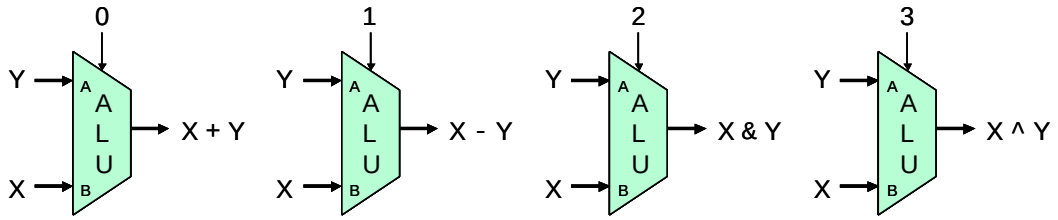
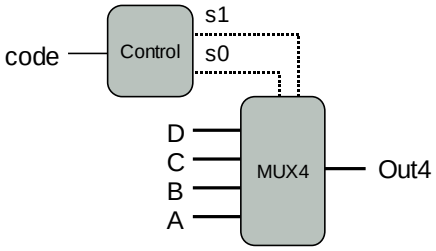


Figure 4.13: **Arithmetic/logic unit (ALU).** Depending on the setting of the function input, the circuit will perform one of four different arithmetic and logical operations.

4.2.4 Set Membership

In our processor designs, we will find many examples where we want to compare one signal against a number of possible matching signals, such as to test whether the code for some instruction being processed matches some category of instruction codes. As a simple example, suppose we want to generate the signals s1 and s0 for the four-way multiplexor of Figure 4.12 by selecting the high- and low-order bits from a two-bit signal code, as follows:



In this circuit, the two-bit signal code would then control the selection among the four data words A, B, C, and D. We can express the generation of signals s1 and s0 using equality tests based on the possible values of code:

```
bool s1 = code == 2 || code == 3;

bool s0 = code == 1 || code == 3;
```

A more concise expression can be written that expresses the property that s1 is 1 when code is in the set $\{2, 3\}$, and s0 is 1 when code is in the set $\{1, 3\}$:

```
bool s1 = code in { 2, 3 };

bool s0 = code in { 1, 3 };
```

The general form of a set membership test is

$$iexpr \quad \texttt{in} \quad \{iexpr_1, iexpr_2, \ldots, iexpr_k\}$$

where both the value being tested, $expr$, and the candidate matches, $iexpr_1$ through $iexpr_k$, are all integer expressions.

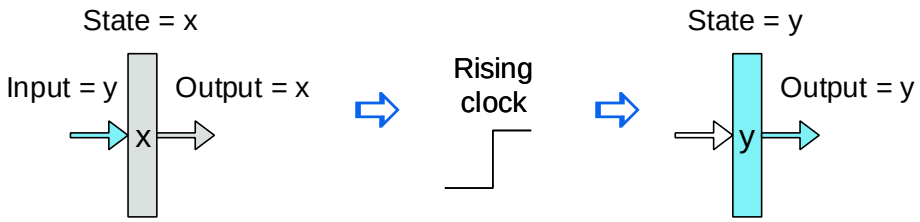


Figure 4.14: **Register operation.** The register outputs remain held at the current register state until the clock signal rises. When the clock rises, the values at the register inputs are captured to become the new register state.

4.2.5 Memory and Clocking

Combinational circuits, by their very nature, do not store any information. Instead, they simply react to the signals at their inputs, generating outputs equal to some function of the inputs. To create *sequential circuits*, that is, systems that have state and perform computations on that state, we must introduce devices that store information represented as bits. We consider two classes of memory devices:

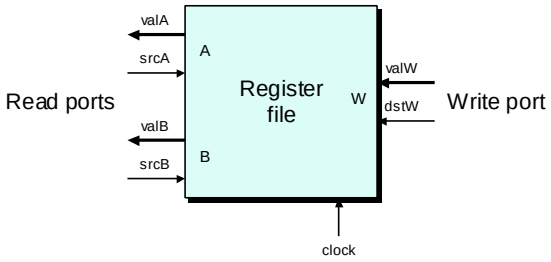
Clocked registers (or simply *registers*) store individual bits or words. A clock signal controls the loading of the register with the value at its input.

Random-access memories (or simply *memories*) store multiple words, using an address to select which word should be read or written. Examples of random-access memories include (1) the virtual memory system of a processor, where a combination of hardware and operating system software make it appear to a processor that it can access any word within a large address space; and (2) the register file, where register identifiers serve as the addresses. In an IA32 or Y86 processor, the register file holds the eight program registers (`%eax`, `%ecx`, etc.).

As we can see, the word "register" means two slightly different things when speaking of hardware versus machine-language programming. In hardware, a register is directly connected to the rest of the circuit by its input and output wires. In machine-level programming, the registers represent a small collection of addressable words in the CPU, where the addresses consist of register IDs. These words are generally stored in the register file, although we will see that the hardware can sometimes pass a word directly from one instruction to another to avoid the delay of first writing and then reading the register file. When necessary to avoid ambiguity, we will call the two classes of registers "hardware registers" and "program registers," respectively.

Figure 4.14 gives a more detailed view of a hardware register and how it operates. For most of the time, the register remains in a fixed state (shown as x), generating an output equal to its current state. Signals propagate through the combinational logic preceding the register, creating a new value for the register input (shown as y), but the register output remains fixed as long as the clock is low. As the clock rises, the input signals are loaded into the register as its next state (y), and this becomes the new register output until the next rising clock edge. A key point is that the registers serve as barriers between the combinational logic in different parts of the circuit. Values only propagate from a register input to its output once every clock cycle at the rising clock edge.

The following diagram shows a typical register file:



This register file has two *read ports*, named A and B, and one *write port*, named W. Such a *multiported* random-access memory allows multiple read and write operations to take place simultaneously. In the register file diagrammed, the circuit can read the values of two program registers and update the state of a third. Each port has an address input, indicating which program register should be selected, and a data output or input giving a value for that program register. The addresses are register identifiers, using the encoding shown in Figure 4.4. The two read ports have address inputs srcA and srcB (short for "source A" and "source B") and data outputs valA and valB (short for "value A" and "value B"). The write port has address input dstW (short for "destination W"), and data input valW (short for "value W").

Although the register file is not a combinational circuit (since it has internal storage), reading words from it operates in the same manner as a block of combinational logic having the addresses as inputs and the data as outputs. When either srcA or srcB is set to some register ID, then after some delay, the value stored in the corresponding program register will appear on either valA or valB. For example, setting srcA to 3 will cause the value of program register %ebx to be read, and this value will appear on output valA.

The writing of words to the register file is controlled by a clock signal in a manner similar to the loading of values into a clocked register. Every time the clock rises, the value on input valW is written to the program register indicated by the register ID on input dstW. When dstW is set to the special ID value 8, no program register is written.

4.3 Sequential Y86 Implementations

Now we have the components required to implement a Y86 processor. As a first step, we describe a processor called SEQ (for "sequential" processor). On each clock cycle, SEQ performs all the steps required to process a complete instruction. This would require a very long cycle time, however, and so the clock rate would be unacceptably low. Our purpose in developing SEQ is to provide a first step toward our ultimate goal of implementing an efficient, pipelined processor.

4.3.1 Organizing Processing into Stages

In general, processing an instruction involves a number of operations. We organize them in a particular sequence of stages, attempting to make all instructions follow a uniform sequence, even though the instructions differ greatly in their actions. The detailed processing at each step depends on the particular instruction being executed. Creating this framework will allow us to design a processor that makes best use