```
        add     $t1, $t1,$t1        # reg $t1 = j * 4
        add     $t2, $a0,$t1        # reg $t2 = v + (j * 4)
        lw      $t3, 0($t2)         # reg $t3   = v[j]
        lw      $t4, 4($t2)         # reg $t4   = v[j + 1]
        slt     $t0, $t4, $t3       #  reg $t0 = 0 if $t4   ≥ $t3
        beq     $t0, $zero,exit2    # go to exit2 if $t4   ≥ $t3
                ...
                (body of second for loop)
                ...
        addi    $s1, $s1, -1        # j = j - 1
        j       for2tst             # jump to test of inner loop
exit2:
```

The Procedure Call in `sort`

The next step is the body of the second *for* loop:

```
swap(v,j);
```

Calling `swap` is easy enough:

```
jal     swap
```

Passing Parameters in `sort`

The problem comes when we want to pass parameters because the `sort` procedure needs the values in registers `$a0` and `$a1`, yet the `swap` procedure needs to have its parameters placed in those same registers. One solution is to copy the parameters for `sort` into other registers earlier in the procedure, making registers `$a0` and `$a1` available for the call of `swap`. (This copy is faster than saving and restoring on the stack.) We first copy `$a0` and `$a1` into `$s2` and `$s3` during the procedure:

```
move    $s2, $a0    # copy parameter $a0 into $s2
move    $s3, $a1    # copy parameter $a1 into $s3
```

Then we pass the parameters to `swap` with these two instructions:

```
move    $a0, $s2    # first swap parameter is v
move    $a1, $s1    # second swap parameter is j
```

Preserving Registers in `sort`

The only remaining code is the saving and restoring of registers. Clearly we must save the return address in register `$ra`, since `sort` is a procedure and is called itself. The `sort` procedure also uses the saved registers `$s0`, `$s1`, `$s2`, and `$s3`, so they must be saved. The prologue of the `sort` procedure is then

```
addi    $sp,$sp,-20  # make room on stack for 5 regs
sw      $ra,16($sp)  # save $ra on stack
sw      $s3,12($sp)  # save $s3 on stack
sw      $s2, 8($sp)  # save $s2 on stack
sw      $s1, 4($sp)  # save $s1 on stack
sw      $s0, 0($sp)  # save $s0 on stack
```

The tail of the procedure simply reverses all these instructions, then adds a `jr` to return.

The Full Procedure `sort`

Now we put all the pieces together in Figure 3.26, being careful to replace references to registers `$a0` and `$a1` in the *for* loops with references to registers `$s2` and `$s3`. Once again to make the code easier to follow, we identify each block of code with its purpose in the procedure. In this example, 9 lines of the `sort` procedure in C became the 35 lines in the MIPS assembly language.

Elaboration: One optimization that would work well in this example is *procedure inlining*. Instead of passing arguments in parameters and invoking the code with a `jal` instruction, the compiler would copy the code from the body of the `swap` procedure where the call to `swap` appears in the code. Inlining would avoid four instructions in this example. The downside of the inlining optimization is that the compiled code would be bigger, assuming that the inlined procedure is called from several locations. Such a code expansion might turn into *lower* performance if it increased the cache miss rate; see Chapter 7.

The MIPS compilers always save room on the stack for the arguments in case they need to be stored, so in reality they always decrement `$sp` by 16 to make room for all 4 argument registers (16 bytes). One reason is that C provides a `vararg` option that allows a pointer to pick, say, the third argument to a procedure. When the compiler encounters the rare `vararg`, it copies the registers onto the stack into the reserved locations.

Saving registers				
	sort:	addi	$sp,$sp, -20	# make room on stack for 5 registers
		sw	$ra, 16($sp)	# save $ra on stack
		sw	$s3,12($sp)	# save $s3 on stack
		sw	$s2, 8($sp)	# save $s2 on stack
		sw	$s1, 4($sp)	# save $s1 on stack
		sw	$s0, 0($sp)	# save $s0 on stack
Procedure body				
Move parameters		move	$s2, $a0	# copy parameter $a0 into $s2 (save $a0)
		move	$s3, $a1	# copy parameter $a1 into $s3 (save $a1)
Outer loop		move	$s0, $zero	# i = 0
	for1tst:	slt	$t0, $s0, $s3	# reg $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
		beq	$t0, $zero, exit1	# go to exit1 if $s0 ≥ $s3 (i ≥ n)
Inner loop		addi	$s1, $s0, -1	# j = i - 1
	for2tst:	slti	$t0, $s1, 0	# reg $t0 = 1 if $s1 < 0 (j < 0)
		bne	$t0, $zero, exit2	# go to exit2 if $s1 < 0 (j < 0)
		add	$t1, $s1, $s1	# reg $t1 = j * 2
		add	$t1, $t1, $t1	# reg $t1 = j * 4
		add	$t2, $s2, $t1	# reg $t2 = v + (j * 4)
		lw	$t3, 0($t2)	# reg $t3 = v[j]
		lw	$t4, 4($t2)	# reg $t4 = v[j + 1]
		slt	$t0, $t4, $t3	# reg $t0 = 0 if $t4 ≥ $t3
		beq	$t0, $zero, exit2	# go to exit2 if $t4 ≥ $t3
Pass parameters and call		move	$a0, $s2	# 1st parameter of swap is v (old $a0)
		move	$a1, $s1	# 2nd parameter of swap is j
		jal	swap	# swap code shown in Figure 3.24
Inner loop		addi	$s1, $s1, -1	# j = j - 1
		j	for2tst	# jump to test of inner loop
Outer loop	exit2:	addi	$s0, $s0, 1	# i = i + 1
		j	for1tst	# jump to test of outer loop
Restoring registers				
	exit1:	lw	$s0, 0($sp)	# restore $s0 from stack
		lw	$s1, 4($sp)	# restore $s1 from stack
		lw	$s2, 8($sp)	# restore $s2 from stack
		lw	$s3,12($sp)	# restore $s3 from stack
		lw	$ra,16($sp)	# restore $ra from stack
		addi	$sp,$sp, 20	# restore stack pointer
Procedure return				
		jr	$ra	# return to calling routine

FIGURE 3.26 MIPS assembly version of procedure `sort` in Figure 3.25 on page 166.

3.11 Arrays versus Pointers

A challenging topic for any new programmer is understanding pointers. Comparing assembly code that uses arrays and array indices to the assembly code that uses pointers offers insight into that difference. This section shows C and MIPS assembly versions of two procedures to clear a sequence of words in memory: one using array indices and one using pointers. Figure 3.27 shows the two C procedures.

Hardware Software Interface

People used to be taught to use pointers in C to get greater efficiency than available with arrays: "Use pointers, even if you can't understand the code." The procedure `clear2` in Figure 3.27 is such an example. Modern optimizing compilers can produce just as good code for the array version of the code. The purpose of this section is to show how pointers map into MIPS instructions, and not to endorse a questionable style.

```
clear1(int array[], int size)
{
  int i;
  for (i = 0; i < size; i = i + 1)
       array[i] = 0;
}

clear2(int *array, int size)
{
  int *p;
  for (p = &array[0]; p < &array[size]; p = p + 1)
       *p = 0;
}
```

FIGURE 3.27 Two C procedures for setting an array to all zeros. `Clear1` uses indices, while `clear2` uses pointers. The second procedure needs some explanation for those unfamiliar with C. The address of a variable is indicated by & and referring to the object pointed to by a pointer is indicated by *. The declarations declare that `array` and `p` are pointers to integers. The first part of the *for* loop in `clear2` assigns the address of the first element of `array` to the pointer `p`. The second part of the *for* loop tests to see if the pointer is pointing beyond the last element of `array`. Incrementing a pointer by one, in the last part of the *for* loop, means moving the pointer to the next sequential object of its declared size. Since `p` is a pointer to integers, the compiler will generate MIPS instructions to increment `p` by four, the number of bytes in a MIPS integer. The assignment in the loop places 0 in the object pointed to by `p`.

Array Version of Clear

Let's start with the array version, `clear1`, focusing on the body of the loop and ignoring the procedure linkage code. We assume that the two parameters `array` and `size` are found in the registers `$a0` and `$a1`, and that `i` is allocated to register `$t0`.

The initialization of `i`, the first part of the *for* loop, is straightforward:

```
        move  $t0,$zero         # i = 0 (register $t0 = 0)
```

To set `array[i]` to 0 we must first get its address. Start by multiplying `i` by 4 to get the byte address:

```
loop1: add   $t1,$t0,$t0        # $t1 = i * 2
       add   $t1,$t1,$t1        # $t1 = i * 4
```

Since the starting address of the array is in a register, we must add it to the index to get the address of `array[i]` using an add instruction:

```
       add   $t2,$a0,$t1        # $t2 = address of array[i]
```

(This example is an ideal situation for indexed addressing; see page 175.) Finally we can store 0 in that address:

```
       sw    $zero, 0($t2)     # array[i] = 0
```

This instruction is the end of the body of the loop, so the next step is to increment `i`:

```
       addi  $t0,$t0,1         # i = i + 1
```

The loop test checks if `i` is less than `size`:

```
       slt   $t3,$t0,$a1       # $t3 = (i < size)
       bne   $t3,$zero,loop1   # if (i < size) go to loop1
```

We have now seen all the pieces of the procedure. Here is the MIPS code for clearing an array using indices:

```
       move  $t0,$zero         # i = 0
loop1: add   $t1,$t0,$t0       # $t1 = i * 2
       add   $t1,$t1,$t1       # $t1 = i * 4
       add   $t2,$a0,$t1       # $t2 = address of array[i]
       sw    $zero, 0($t2)     # array[i] = 0
       addi  $t0,$t0,1         # i = i + 1
       slt   $t3,$t0,$a1       # $t3 = (i < size)
       bne   $t3,$zero,loop1   # if (i < size) go to loop1
```

(This code works as long as `size` is greater than 0.)

Pointer Version of Clear

The second procedure that uses pointers allocates the two parameters `array` and `size` to the registers `$a0` and `$a1` and allocates `p` to register `$t0`. The code for the second procedure starts with assigning the pointer `p` to the address of the first element of the array:

```
        move    $t0,$a0       # p = address of array[0]
```

The next code is the body of the *for* loop, which simply stores 0 into `p`:

```
loop2:  sw      $zero,0($t0) # Memory[p] = 0
```

This instruction implements the body of the loop, so the next code is the iteration increment, which changes `p` to point to the next word:

```
        addi    $t0,$t0,4     # p = p + 4
```

Incrementing a pointer by 1 means moving the pointer to the next sequential object in C. Since `p` is a pointer to integers, each of which use 4 bytes, the compiler increments `p` by 4.

The loop test is next. The first step is calculating the address of the last element of `array`. Start with multiplying `size` by 4 to get its byte address:

```
        add   $t1,$a1,$a1     # $t1 = size * 2
        add   $t1,$t1,$t1     # $t1 = size * 4
```

and then we add the product to the starting address of the array to get the address of the first word *after* the array:

```
        add   $t2,$a0,$t1     # $t2 = address of array[size]
```

The loop test is simply to see if `p` is less than the last element of `array`:

```
        slt   $t3,$t0,$t2     # $t3 = (p<&array[size])
        bne   $t3,$zero,loop2# if (p<&array[size]) go to loop2
```

With all the pieces completed, we can show a pointer version of the code to zero an array:

```
        move $t0,$a0          # p = address of array[0]
loop2:  sw   $zero,0($t0)     # Memory[p] = 0
        addi $t0,$t0,4        # p = p + 4
        add  $t1,$a1,$a1      # $t1 = size * 2
        add  $t1,$t1,$t1      # $t1 = size * 4
        add  $t2,$a0,$t1      # $t2 = address of array[size]
        slt  $t3,$t0,$t2      # $t3 = (p<&array[size])
        bne  $t3,$zero,loop2# if (p<&array[size]) go to loop2
```

As in the first example, this code assumes `size` is greater than 0.

Note that this program calculates the address of the end of the array every iteration of the loop, even though it does not change. A faster version of the code moves this calculation outside the loop:

```
        move $t0,$a0          # p = address of array[0]
        add  $t1,$a1,$a1      # $t1 = size * 2
        add  $t1,$t1,$t1      # $t1 = size * 4
        add  $t2,$a0,$t1      # $t2 = address of array[size]
loop2:  sw   $zero,0($t0)     # Memory[p] = 0
        addi $t0,$t0,4        # p = p + 4
        slt  $t3,$t0,$t2      # $t3 = (p<&array[size])
        bne  $t3,$zero,loop2 # if (p<&array[size]) go to loop2
```

Comparing the Two Versions of Clear

Comparing the two code sequences side by side illustrates the difference between array indices and pointers (the changes introduced by the pointer version are highlighted):

```
        move $t0,$zero    # i = 0
loop1:  add  $t1,$t0,$t0  # $t1 = i * 2
        add  $t1,$t1,$t1  # $t1 = i * 4
        add  $t2,$a0,$t1  # $t2 = &array[i]
        sw   $zero, 0($t2)# array[i] = 0
        addi $t0,$t0,1    # i = i + 1
        slt  $t3,$t0,$a1  # $t3 = (i < size)
        bne  $t3,$zero,loop1# if () go to loop1
```

```
        move $t0,$a0     # p = & array[0]
        add  $t1,$a1,$a1 # $t1 = size * 2
        add  $t1,$t1,$t1 # $t1 = size * 4
        add  $t2,$a0,$t1 # $t2 = &array[size]
loop2:  sw   $zero,0($t0)# Memory[p] = 0
        addi $t0,$t0,4   # p = p + 4
        slt  $t3,$t0,$t2 # $t3=(p<&array[size])
        bne  $t3,$zero,loop2# if () go to loop2
```

The version on the left must have the "multiply" and add inside the loop because `i` is incremented and each address must be recalculated from the new index; the memory pointer version on the right increments the pointer `p` directly. The pointer version reduces the instructions executed per iteration from 7 to 4. Many modern compilers will optimize the C code in `clear1` to produce code similar to the assembly code above on the right-hand side.

Elaboration: The C compiler would add a test to be sure that size is greater than 0. One way would be to add a jump just before the first instruction of the loop to the `slt` instruction.

3.12 Real Stuff: PowerPC and 80x86 Instructions

Beauty is altogether in the eye of the beholder.

Margaret Wolfe Hungerford, *Molly Bawn*, 1877

Designers of instruction sets sometimes provide more powerful operations than those found in MIPS. The goal is generally to reduce the number of instructions executed by a program. The danger is that this reduction can occur at the cost of simplicity, increasing the time a program takes to execute because the instructions are slower. This slowness may be the result of a slower clock cycle time or of requiring more clock cycles than a simpler sequence (see section 2.8 on page 82).

The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions. Section 3.13 demonstrates the pitfalls of complexity.

The IBM/Motorola PowerPC

The PowerPC, made by IBM and Motorola and used in the Apple Macintosh, shares many similarities to MIPS: both have 32 integer registers, instructions are all 32 bits long, and data transfer is possible only with loads and stores. The primary difference is two more addressing modes plus a few operations.

Indexed Addressing

In the examples above we saw cases where we needed one register to hold the base of the array and the other to hold the index of the array. PowerPC provides an addressing mode, often called *indexed addressing*, that allows two registers to be added together. The MIPS code

```
add   $t0,$a0,$s3  # $a0 has base of an array, $s3 is index
lw    $t1,0($t0)   # reg $t1 gets Memory[$a0+$s3]
```

could be replaced by the following single instruction in PowerPC:

```
lw    $t1,$a0+$s3  # reg $t1 gets Memory[$a0+$s3]
```

Using the same notation as Figure 3.17, Figure 3.28 shows indexed addressing. It is available with both loads and stores.

Update Addressing

Imagine the case of a code sequence marching through an array of words in memory, such as in the array version of `clear1` on page 172. A frequent pair

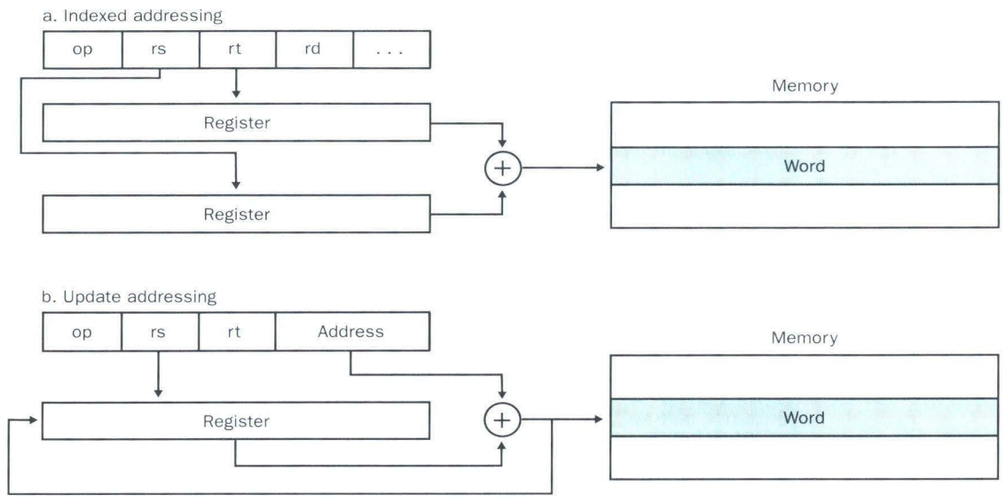


FIGURE 3.28 Illustration of indexed and update addressing mode. The operand is shaded in color.

of operations would be loading a word and then incrementing the base register to point to the next word. The idea of *update addressing* is to have a new version of data transfer instructions that will automatically increment the base register to point to the next word each time data is transferred. Since the MIPS architecture uses byte addresses and words are 4 bytes, this new form would be equivalent to this pair of MIPS instructions:

```
lw    $t0,4($s3)   # reg $t0 gets Memory[$s3+4]
addi  $s3,$s3,4    # $s3 = $s3 + 4
```

The PowerPC includes an instruction like this:

```
lwu   $t0,4($s3)   # reg $t0=Memory[$s3+4]; $s3 = $s3+4
```

That is, the register is updated with the address calculated as part of the load. Figure 3.28 also shows update addressing. PowerPC has update addressing options for both base and indexed addressing, and for both loads and stores.

Unique PowerPC Instructions

The PowerPC instructions follow the same architecture style as MIPS, largely relying on fast execution of simple instructions for performance. Here are a few exceptions.

The first is load multiple and store multiple. These can transfer up to 32 words of data in a single instruction and are intended to make fast copies of locations in memory by using load multiple and store multiple back to back. They also save code size when saving or restoring registers.

A second example is loops. The PowerPC has a special counter register, separate from the other 32 registers, to try to improve performance of a *for* loop.

Suppose we wanted to execute the following C code:

```
for (i = n; i != 0; i = i - 1)
  {. . .};
```

If we want to decrement a register, compare to 0, and then branch as long as the register is not 0, we could use the following MIPS instructions:

```
Loop: ...
      addi   $t0,$t0,-1       # $t0 = $t0 - 1
      bne    $t0,$zero, Loop  # if $t0 != 0 go to Loop
```

In PowerPC we could use a single instruction instead:

```
      bc     Loop,$ctr!=0     # $ctr = $ctr - 1;
                              # if $ctr != 0 go to Loop
```

Hardware Software Interface

In addition to going against the advice of simplicity, such sophisticated operations may not *exactly* match what the compiler needs to produce. For example, suppose that instead of decrementing by one, the compiler wanted to increment by four, or instead of branching on not equal zero, the compiler wanted to branch if the index was less than or equal to the limit. Then the instruction just described would be a mismatch. When faced with such objections, the instruction set designer might then generalize the operation, adding another operand to specify the increment and perhaps an option on which branch condition to use. Then the danger is that a common case, say, incrementing by one, will be slower than a sequence of simple operations.

The Intel 80x86

MIPS was the vision of a single small group in 1985; the pieces of this architecture fit nicely together, and the whole architecture can be described succinctly. Such is not the case for the 80x86; it is the product of several independent groups who evolved the architecture over almost 20 years, adding new features to the original instruction set as someone might add clothing to a packed bag. Here are important 80x86 milestones:

- **1978**: The Intel 8086 architecture was announced as an assembly-language-compatible extension of the then-successful Intel 8080, an 8-bit microprocessor. The 8086 is a 16-bit architecture, with all internal registers 16 bits wide. Unlike MIPS, the registers have dedicated uses, and hence the 8086 is not considered a *general-purpose register* architecture.
- **1980**: The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Instead of using registers, it relies on a stack (see section 3.15 and section 4.9).
- **1982**: The 80286 extended the 8086 architecture by increasing the address space to 24 bits, by creating an elaborate memory-mapping and protection model (see Chapter 7), and by adding a few instructions to round out the instruction set and to manipulate the protection model.
- **1985**: The 80386 extended the 80286 architecture to 32 bits. In addition to a 32-bit architecture with 32-bit registers and a 32-bit address space, the 80386 added new addressing modes and additional operations. The added instructions make the 80386 nearly a general-purpose register machine. The 80386 also added paging support in addition to segmented addressing (see Chapter 7). Like the 80286, the 80386 has a mode to execute 8086 programs without change.
- **1989–95**: The subsequent 80486 in 1989, Pentium in 1992, and Pentium Pro in 1995 were aimed at higher performance, with only four instructions added to the user-visible instruction set: three to help with multiprocessing (Chapter 9) and a conditional move instruction.
- **1997**: After the Pentium and Pentium Pro were shipping, Intel announced that it would expand the Pentium and the Pentium Pro architectures with MMX. This new set of 57 instructions uses the floating-point stack to accelerate multimedia and communication applications. MMX instructions typically operate on multiple short data elements at a time, in the tradition of single instruction, multiple data (SIMD) architectures (see Chapter 9).

This history illustrates the impact of the "golden handcuffs" of compatibility on the 80x86, as the existing software base at each step was too important to jeopardize with significant architectural changes.

Whatever the artistic failures of the 80x86, keep in mind that there are more instances of this architectural family than of any other in the world, perhaps 300 million in 1997. Nevertheless, this checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

Brace yourself for what you are about to see! Do *not* try to read this section with the care you would need to write 80x86 programs; the goal instead is to give you familiarity with the strengths and weaknesses of the world's most popular architecture.

Rather than show the entire 16-bit and 32-bit instruction set, in this section we concentrate on the 32-bit subset that originated with the 80386, as this portion of the architecture will be increasingly dominant over time. We start our explanation with the registers and addressing modes, move on to the integer operations, and conclude with an examination of instruction encoding.

80x86 Registers and Data Addressing Modes

The evolution of the instruction set can be seen in the registers of the 80386 (Figure 3.29). The 80386 basically extended all 16-bit registers (except the segment registers) to 32 bits, prefixing an *E* to their name to indicate the 32-bit version. We'll refer to them generically as GPRs (general-purpose registers). The 80386 contains only eight GPRs. This means MIPS programs can use four times as many.

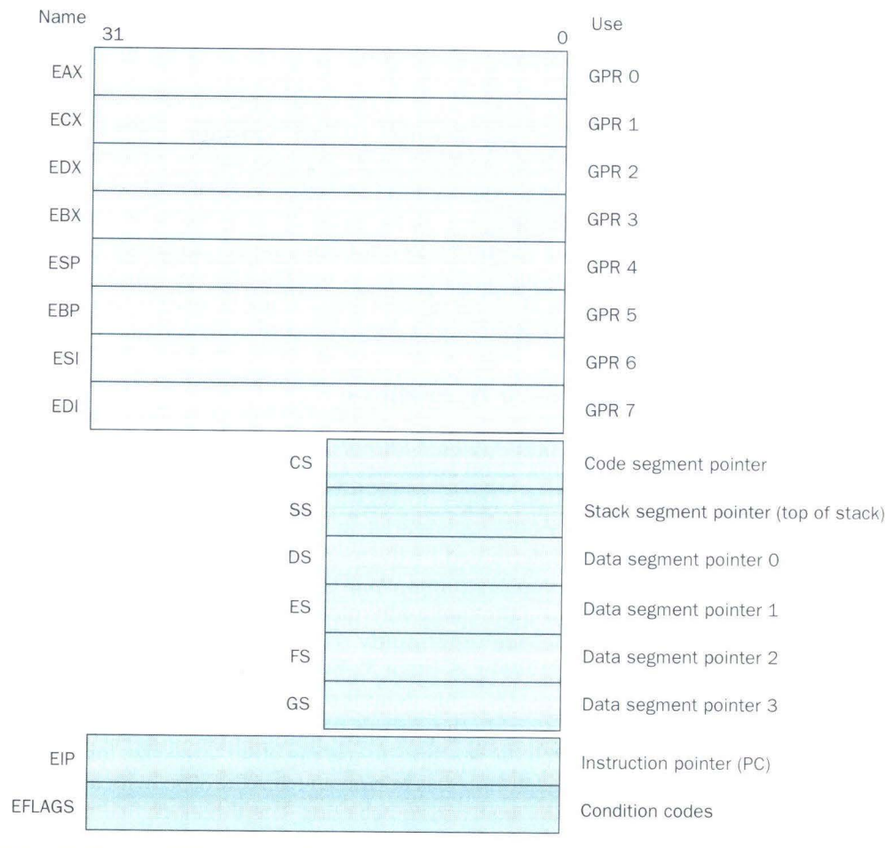


FIGURE 3.29 The 80386 register set. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers.

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

FIGURE 3.30 Instruction types for the arithmetic, logical, and data transfer instructions. The 80x86 allows the combinations shown. The only restriction is the absence of a memory-memory mode. Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure 3.29 (not EIP or EFLAGS).

The arithmetic, logical, and data transfer instructions are two-operand instructions that allow the combinations shown in Figure 3.30. There are two important differences here. The 80x86 arithmetic and logical instructions must have one operand act as both a source and a destination; MIPS allows separate registers for source and destination. This restriction puts more pressure on the limited registers, since one source register must be modified. The second important difference is that one of the operands can be in memory. Thus virtually any instruction may have one operand in memory, unlike MIPS and PowerPC.

The seven data memory-addressing modes, described in detail below, offer two sizes of addresses within the instruction. These so-called *displacements* can be 8 bits or 32 bits.

Although a memory operand can use any addressing mode, there are restrictions on which *registers* can be used in a mode. Figure 3.31 shows the 80x86 addressing modes and which GPRs cannot be used with that mode, plus how you would get the same effect using MIPS instructions.

80x86 Integer Operations

The 8086 provides support for both 8-bit (*byte*) and 16-bit (*word*) data types. The 80386 adds 32-bit addresses and data (*double words*) in the 80x86. The data type distinctions apply to register operations as well as memory accesses. Almost every operation works on both 8-bit data and on one longer data size. That size is determined by the mode, and is either 16 bits or 32 bits.

Clearly some programs want to operate on data of all three sizes, so the 80386 architects provide a convenient way to specify each version without expanding code size significantly. They decided that most programs would be dominated by either 16-bit or 32-bit data, and so it made sense to be able to set a default large size. This default data size is set by a bit in the code segment register. To override the default data size, an 8-bit *prefix* is attached to the instruction to tell the machine to use the other large size for this instruction.

The prefix solution was borrowed from the 8086, which allows multiple prefixes to modify instruction behavior. The three original prefixes override the

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	`lw $s0,100($s1) # ≤16-bit # displacement`

Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($t0)# ≤16-bit # displacement`

FIGURE 3.31 80x86 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 3.24 and 3.26). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

default segment register, lock the bus to support a semaphore (see Chapter 9), or repeat the following instruction until the register ECX counts down to 0. This last prefix was intended to be paired with a byte move instruction to move a variable number of bytes. The 80386 also added a prefix to override the default address size.

The 80x86 integer operations can be divided into four major classes:

1. Data movement instructions, including move, push, and pop
2. Arithmetic and logic instructions, including test and integer and decimal arithmetic operations
3. Control flow, including conditional branches, unconditional jumps, calls, and returns
4. String instructions, including string move and string compare

The first two categories are unremarkable, except that the arithmetic and logic instruction operations allow the destination to either be a register or a memory location. Figure 3.32 shows some typical 80x86 instructions and their functions.

Conditional branches on the PowerPC and the 80x86 are based on *condition codes* or *flags*. Condition codes are set as a side effect of an operation; most are used to compare the value of a result to 0. Branches then test the condition codes. The argument for condition codes is that they occur as part of normal operations and are faster to test than it is to compare registers as MIPS does for

Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42_{hex}
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

FIGURE 3.32 Some typical 80x86 instructions and their functions. A list of frequent operations appears in Figure 3.33. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

beq and bne. The argument against condition codes is that the compare to 0 extends the time of the operation, since it uses extra hardware after the operation, and that often the programmer must use compare instructions to test a value that is not the result of an operation. Also, PC-relative branch addresses must be specified in the number of bytes, since unlike MIPS, 80386 instructions are not all 4 bytes in length.

String instructions are part of the 8080 ancestry of the 80x86 and are not commonly executed in most programs. They are often slower than equivalent software routines (see the fallacy on page 185).

Figure 3.33 lists some of the integer 80x86 instructions. Many of the instructions are available in both byte and word formats.

80x86 Instruction Encoding

Saving the worst for last, the encoding of instructions in the 8086 is complex, with many different instruction formats. Instructions for the 80386 may vary from 1 byte, when there are no operands, up to 17 bytes.

Figure 3.34 shows the instruction format for several of the example instructions in Figure 3.32. The opcode byte usually contains a bit saying whether the operand is 8 bits or 32 bits. For some instructions the opcode may include the addressing mode and the register; this is true in many instructions that have the form "register = register op immediate." Other instructions use a "postbyte" or extra opcode byte, labeled "mod, reg, r/m," which contains the addressing mode information. This postbyte is used for many of the instructions that address memory. The base plus scaled index mode uses a second postbyte, labeled "sc, index, base."

Instruction	Meaning
Control	**Conditional and unconditional branches**
JNZ, JZ	Jump if condition to EIP + 8-bit offset; JNE (for JNZ), JE (for JZ) are alternative names
JMP	Unconditional jump—8-bit or 16-bit offset
CALL	Subroutine call—16-bit offset; return address pushed onto stack
RET	Pops return address from stack and jumps to it
LOOP	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX ≠ 0
Data transfer	**Move data between registers or between register and memory**
MOV	Move between two registers or between register and memory
PUSH, POP	Push source operand on stack; pop operand from stack top to a register
LES	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
ADD, SUB	Add source to destination; subtract source from destination; register-memory format
CMP	Compare source and destination; register-memory format
SHL, SHR, RCR	Shift left; shift logical right; rotate right with carry condition code as fill
CBW	Convert byte in 8 rightmost bits of EAX to 16-bit word in right of EAX
TEST	Logical AND of source and destination sets condition codes
INC, DEC	Increment destination, decrement destination; register-memory format
OR, XOR	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
MOVS	Copies from string source to destination by incrementing ESI and EDI; may be repeated
LODS	Loads a byte, word, or double word of a string into the EAX register

FIGURE 3.33 Some typical operations on the 80x86. Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

Figure 3.35 shows the encoding of the two postbyte address specifiers for both 16-bit and 32-bit mode. Unfortunately, to fully understand which registers and which addressing modes are available, you need to see the encoding of all addressing modes and sometimes even the encoding of the instructions.

80x86 Conclusion

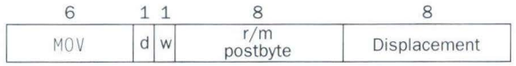
Intel had a 16-bit microprocessor two years before its competitors' more elegant architectures, such as the Motorola 68000, and this head start led to the selection of the 8086 as the CPU for the IBM PC. Intel engineers generally acknowledge that the 80x86 is more difficult to build than machines like MIPS, but the much larger market means Intel can afford more resources to help overcome the added complexity. What the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective.

The saving grace is that the most frequently used 80x86 architectural components are not too difficult to implement, as Intel has demonstrated by rapidly improving performance of integer programs since 1978. To get that performance, compilers must avoid the portions of the architecture that are hard to implement fast.

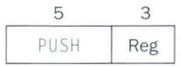
a. JE EIP + displacement

b. CALL

c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765

f. TEST EDX, #42

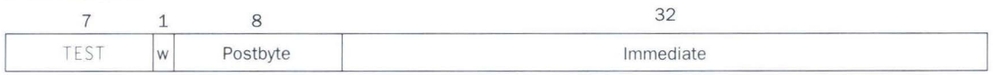


FIGURE 3.34 Typical 80x86 instruction formats. The encoding of the postbyte is shown in Figure 3.35. Many instructions contain the 1-bit field w, which says whether the operation is a byte or double word. The d field in MOV is used in instructions that may move to or from memory and shows the direction of the move. The ADD instruction requires 32 bits for the immediate field because in 32-bit mode the immediates are either 8 bits or 32 bits. The immediate field in the TEST is 32 bits long because there is no 8-bit immediate for test in 32-bit mode. Overall, instructions may vary from 1 to 17 bytes in length. The long length comes from extra 1-byte prefixes, having both a 4-byte immediate and a 4-byte displacement address, using an opcode of 2 bytes, and using the scaled index mode specifier, which adds another byte.

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

FIGURE 3.35 The encoding of the first address specifier of the 80x86, *"mod, reg, r/m."* The first four columns show the encoding of the 3-bit reg field, which depends on the w bit from the opcode and whether the machine is in 16-bit mode (8086) or 32-bit mode (80386). The remaining columns explain the mod and r/m fields. The meaning of the 3-bit r/m field depends on the value in the 2-bit mod field and the address size. Basically, the registers used in the address calculation are listed in the sixth and seventh columns, under mod = 0, with mod = 1 adding an 8-bit displacement and mod = 2 adding a 16-bit or 32-bit displacement, depending on the address mode. The exceptions are r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; r/m = 5 when mod = 1 or mod = 2 in 32-bit mode selects EBP plus displacement; and r/m = 4 in 32-bit mode when mod ≠ 3, where (sib) means use the scaled index mode shown in Figure 3.31 on page 181. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

3.13 Fallacies and Pitfalls

Fallacy: More powerful instructions mean higher performance.

Part of the power of the Intel 80x86 is the prefixes that can modify the execution of the following instruction. One prefix can repeat the following instruction until a counter counts down to 0. Thus, to move data in memory, it would seem that the natural instruction sequence is to use move with the repeat prefix to perform 32-bit memory-to-memory moves. On a 133-MHz Pentium (with the Triton chip set, 60-ns EDO DRAM, 256-KB cache), this user-level program can move data at about 40 MB/sec.

An alternative method, which uses the standard instructions found in all computers, is to load the data into the registers and then store the registers back to memory. This second version of this program, with the code replicated so as to reduce loop overhead, copies at about 60 MB/sec on the same machine, or 1.5 times faster. A third version, which used the larger floating-point registers instead of the integer registers of the 80x86, copies at about 80 MB/sec, or 2.0 times faster than the complex instruction.

Fallacy: Write in assembly language to obtain the highest performance.

At one time compilers for programming languages produced naive instruction sequences; the increasing sophistication of compilers means the gap between compiled code and code produced by hand is closing fast. In fact, to compete with current compilers, the assembly language programmer needs to thoroughly understand the concepts in Chapters 6 and 7 on processor pipelining and memory hierarchy.

This battle between compilers and assembly language coders is one situation in which humans are losing ground. For example, C offers the programmer a chance to give a hint to the compiler about which variables should be kept in registers versus spilled to memory. When compilers were poor at register allocation, such hints were vital to performance. In fact, some C textbooks spent a fair amount of time giving examples that effectively use register hints. Today's C compilers generally ignore such hints because the compiler does a better job at allocation than the programmer.

As a specific counterexample, we ran the MIPS assembly language programs in Figures 3.24 and 3.26 to compare performance to the C programs in Figures 3.23 and 3.25. Figure 3.36 shows the results. As you can see, the compiled program is 1.5 times faster than the assembled program. The compiler generally was able to create assembly language code that was tailored exactly to these conditions, while the assembly language program was written in a slightly more general fashion to make it easier to modify and understand. The specific improvements of the C compiler were a more streamlined procedure linkage convention and changing the address calculations to move the multiply outside the inner loop.

Even *if* writing by hand resulted in faster code, the dangers of writing in assembly language are longer time spent coding and debugging, the loss in portability, and the difficulty of maintaining such code. One of the few widely accepted axioms of software engineering is that coding takes longer if you write more lines, and it clearly takes many more lines to write a program in assembly language than in C. And once it is coded, the next danger is that it will become a popular program. Such programs always live longer than expected, meaning that someone will have to update the code over several years and make it work with new releases of operating systems and new models of machines. Writing in higher-level language instead of assembly language not only allows future compilers to tailor the code to future machines, it also makes the software easier to maintain and allows the program to run on more brands of computers.

Language	Time
Assembly	37.9 seconds
C	25.3 seconds

FIGURE 3.36 Performance comparison of the C and assembly language versions of the `sort` and `swap` procedures in section 3.10. The size of the array to be sorted was increased to 10,000 elements. The programs were run on a DECsystem 5900 with 128 MB of main memory and a 40-MHz R3000 processor using version 4.2a (Revision 47) of the Ultrix operating system. The C compiler was run with the –O option.

Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by one.

Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by one instead of by the word size in bytes. Forewarned is forearmed!

Pitfall: Using a pointer to an automatic variable outside its defining procedure.

A common mistake in dealing with pointers is to pass a result from a procedure that includes a pointer to an array that is declared local to that procedure. Following the stack discipline, in Figure 3.12 on page 139, the memory that contains the local array will be reused as soon as the procedure returns. Pointers to automatic variables can lead to chaos.

3.14 Concluding Remarks

Less is more.

Robert Browning, *Andrea del Sarto*, 1855

The two principles of the *stored-program* computer are the use of instructions that are indistinguishable from numbers and the use of alterable memory for programs. These principles allow a single machine to aid environmental scientists, financial advisers, and novelists in their specialties. The selection of a set of instructions that the machine can understand demands a delicate balance among the number of instructions needed to execute a program, the number of clock cycles needed by an instruction, and the speed of the clock. Four design principles guide the authors of instruction sets in making that delicate balance:

1. *Simplicity favors regularity.* Regularity motivates many features of the MIPS instruction set: keeping all instructions a single size, always requiring three register operands in arithmetic instructions, and keeping the register fields in the same place in each instruction format.

2. *Smaller is faster.* The desire for speed is the reason that MIPS has 32 registers rather than many more.
3. *Good design demands good compromises.* One MIPS example was the compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length.
4. *Make the common case fast.* Examples of making the common MIPS case fast include PC-relative addressing for conditional branches and immediate addressing for constant operands.

Above this machine level is assembly language, a language that humans can read. The assembler translates it into the binary numbers that machines can understand, and it even "extends" the instruction set by creating symbolic instructions that aren't in the hardware. For instance, constants or addresses that are too big are broken into properly sized pieces, common variations of instructions are given their own name, and so on. The MIPS instructions we have covered so far (both real and pseudo) are listed in Figure 3.37.

These instructions are not born equal; the popularity of the few dominates the many. For example, Figure 3.38 shows the popularity of each class of instructions for two programs, gcc and spice. The varying popularity of instructions plays an important role in the chapters on performance, datapath, control, and pipelining.

Each category of MIPS instructions is associated with constructs that appear in programming languages:

- The arithmetic instructions correspond to the operations found in assignment statements.
- Data transfer instructions are most likely to occur when dealing with data structures like arrays or structures.
- The conditional branches are used in *if* statements and in loops.
- The unconditional jumps are used in procedure calls and returns and also for *case/switch* statements.

More of the MIPS instruction set is revealed in Chapter 4, after we explain computer arithmetic.

MIPS instructions	Name	Format	Pseudo MIPS	Name	Format
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multi	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load byte	lb	I	branch less than or equal	ble	I
store byte	sb	I	branch greater than	bgt	I
load upper immediate	lui	I	branch greater than or equal	bge	I
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

FIGURE 3.37 The MIPS instruction set covered so far, with the real MIPS instructions on the left and the pseudoinstructions on the right. Appendix A (section A.10 on page A-49) describes the full MIPS architecture. Figure 3.18 on page 153 shows more details of the MIPS architecture revealed in this chapter.

Instruction class	MIPS examples	HLL correspondence	Frequency	
			gcc	spice
Arithmetic	add, sub, addi	operations in assignment statements	48%	50%
Data transfer	lw, sw, lb, sb, lui	references to data structures, such as arrays	33%	41%
Conditional branch	beq, bne, slt, slti	*if* statements and loops	17%	8%
Jump	j, jr, jal	procedure calls, returns, and *case/switch* statements	2%	1%

FIGURE 3.38 MIPS instruction classes, examples, correspondence to high-level program language constructs, and percentage of MIPS instructions executed by category for two programs, gcc and spice. Figure 4.54 on page 311 shows the percentage of the individual MIPS instructions executed.

3.15 Historical Perspective and Further Reading

accumulator: Archaic term for register. On-line use of it as a synonym for "register" is a fairly reliable indication that the user has been around quite a while.

Eric Raymond, *The New Hacker's Dictionary*, 1991

Accumulator Architectures

Hardware was precious in the earliest stored-program computers. As a consequence, computer pioneers could not afford the number of registers found in today's machines. In fact, these machines had a single register for arithmetic instructions. Since all operations would accumulate in a single register, it was called the *accumulator*, and this style of instruction set is given the same name. For example, EDSAC in 1949 had a single accumulator.

The three-operand format of MIPS suggests that a single register is at least two registers shy of our needs. Having the accumulator as both a source operand *and* as the destination of the operation fills part of the shortfall, but it still leaves us one operand short. That final operand is found in memory. Accumulator machines have the memory-based operand-addressing mode suggested earlier. It follows that the add instruction of an accumulator instruction set would look like this:

```
add    200
```

This instruction means add the accumulator to the word in memory at address 200 and place the sum back into the accumulator. No registers are specified because the accumulator is known to be both a source and a destination of the operation.

Compiling an Assignment Statement into Accumulator Instructions

Example What is the accumulator-style assembly code for this C code?

```
A = B + C;
```

Answer It would be translated into the following instructions in an accumulator instruction set:

```
load   AddressB # Acc = Memory[AddressB], or Acc = B
add    AddressC # Acc = B + Memory[AddressC],or Acc = B + C
store  AddressA # Memory[AddressA] = Acc, or A = B + C
```

All variables in a program are allocated to memory in accumulator machines, instead of normally to registers as we saw for MIPS. One way to think about this is that variables are always spilled to memory in this style of machine. As you may imagine, it takes many more instructions to execute a program with a single-accumulator architecture. (See Exercise 3.19 for another example.)

The next step in the evolution of instruction sets was the addition of registers dedicated to specific operations. Hence, registers might be included to act as indices for array references in data transfer instructions, to act as separate accumulators for multiply or divide instructions, and to serve as the top-of-stack pointer. Perhaps the best-known example of this style of instruction set is found in the Intel 8086, the computer at the core of the IBM Personal Computer. This style of instruction set is labeled *extended accumulator, dedicated register,* or *special-purpose register.* Like the single-register accumulator machines, one operand may be in memory for arithmetic instructions. Like the MIPS architecture, however, there are also instructions where all the operands are registers.

General-Purpose Register Architectures

The generalization of the dedicated-register machine allows all the registers to be used for any purpose, hence the name *general-purpose register.* MIPS is an example of a general-purpose register machine. This style of instruction set may be further divided into those that allow one operand to be in memory as found in accumulator machines, called a *register-memory* architecture, and those that demand that operands always be in registers, called either a *load-store* or a *register-register* machine. Figure 3.39 shows a history of the number of registers in some popular computers.

Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	accumulator	1949
IBM 701	1	accumulator	1953
CDC 6600	8	load-store	1963
IBM 360	16	register-memory	1964
DEC PDP-8	1	accumulator	1965
DEC PDP-11	8	register-memory	1970
Intel 8008	1	accumulator	1972
Motorola 6800	2	accumulator	1974
DEC VAX	16	register-memory, memory-memory	1977
Intel 8086	1	extended accumulator	1978
Motorola 68000	16	register-memory	1980
Intel 80386	8	register-memory	1985
MIPS	32	load-store	1985
HP PA-RISC	32	load-store	1986
SPARC	32	load-store	1987
PowerPC	32	load-store	1992
DEC Alpha	32	load-store	1992

FIGURE 3.39 Number of general-purpose registers in popular machines over the years.

The first load-store machine was the CDC 6600 in 1963, considered by many to be the first supercomputer. MIPS is a more recent example of a load-store machine.

The 80386 is Intel's attempt to transform the 80x86 into a general-purpose register-memory instruction set. Perhaps the best-known register-memory instruction set is the IBM 360 architecture, first announced in 1964. This instruction set is still at the core of IBM's mainframe computers—responsible for a large part of the business of the largest computer company in the world. Register-memory architectures were the most popular in the 1960s and the first half of the 1970s.

Digital Equipment Corporation's VAX architecture took memory operands one step further in 1977. It allowed any combination of registers and memory operands to be used in an instruction. A style of machine in which all operands can be in memory is called *memory-memory*. (In truth the VAX instruction set, like almost all other instruction sets since the IBM 360, is a hybrid since it also has general-purpose registers.)

Compiling an Assignment Statement into Memory-Memory Instructions

Example What is the memory-memory style assembly code for this C code?

```
A = B + C;
```

Answer It would be translated into the following instructions in a memory-memory instruction set:

```
add     AddressA,AddressB,AddressC
```

(See Exercise 3.19 for another example.)

Although MIPS has a single add instruction with 32-bit operands, the Intel 80x86 has many versions of a 32-bit add to specify whether an operand is in memory or is in a register. In addition, the memory operand can be accessed with more than seven addressing modes. This combination of address modes and register/memory operands means that there are dozens of variants of an 80x86 add instruction. Clearly this variability makes 80x86 implementations more challenging.

Compact Code and Stack Architectures

When memory is scarce, it is also important to keep programs small, so machines like the Intel 80x86, IBM 360, and VAX had variable-length instructions, both to match the varying operand specifications and to minimize code size. Intel 80x86 instructions are from 1 to 17 bytes long; IBM 360 instructions are 2, 4, or 6 bytes long; and VAX instruction lengths are anywhere from 1 to 54 bytes. If instruction memory space becomes precious once again, such techniques could return to popularity.

In the 1960s, a few companies followed a radical approach to instruction sets. In the belief that it was too hard for compilers to utilize registers effectively, these companies abandoned registers altogether! Instruction sets were based on a *stack model* of execution, like that found in the older Hewlett-Packard handheld calculators. Operands are pushed on the stack from memory or popped off the stack into memory. Operations take their operands from the stack and then place the result back onto the stack. In addition to simplifying compilers by eliminating register allocation, stack machines lent themselves to compact instruction encoding, thereby removing memory size as an excuse not to program in high-level languages.

Compiling an Assignment Statement into Stack Instructions

Example What is the stack-style assembly code for this C code?

```
A = B + C;
```

Answer It would be translated into the following instructions in a stack instruction set:

```
push  AddressC  # Top=Top+4;Stack[Top]=Memory[AddressC]
push  AddressB  # Top=Top+4;Stack[Top]=Memory[AddressB]
add             # Stack[Top-4]=Stack[Top]
                # + Stack[Top-4];Top=Top-4;
pop   AddressA  # Memory[AddressA]=Stack[Top];
                #   Top=Top-4;
```

To get the proper byte address, we adjust the stack by 4. The downside of stacks as compared to registers is that it is hard to reuse data that has been fetched or calculated without repeatedly going to memory. (See Exercise 3.19 for another example.)

Memory space may be precious again for the heralded Network Computer (NC), both because memory space is limited to keep costs low and because programs must be downloaded over the Internet, and smaller programs take less time to transmit. Hence compactness in instruction set encoding is desired for the NC. Such arguments have been used to justify building a hardware interpreter for the Java intermediate language, which is based on a stack. Time will tell whether these arguments have technical versus marketing merit.

High-Level-Language Computer Architectures

In the 1960s, systems software was rarely written in high-level languages. For example, virtually every commercial operating system before Unix was programmed in assembly language, and more recently even OS/2 was originally programmed at that same low level. Some people blamed the code density of the instruction sets rather than the programming languages and the compiler technology.

Hence a machine-design philosophy called *high-level-language computer architecture* was advocated, with the goal of making the hardware more like the programming languages. More efficient programming languages and compilers, plus expanding memory, doomed this movement to a historical footnote. The Burroughs B5000 was the commercial fountainhead of this philosophy, but today there is no significant commercial descendent of this 1960s radical.

Reduced Instruction Set Computer Architectures

This language-oriented design philosophy was replaced in the 1980s by *RISC (reduced instruction set computer)*. Improvements in programming languages, compiler technology, and memory cost meant that less programming was being done at the assembly level, so instruction sets could be measured by how well compilers used them as opposed to how well assembly language programmers used them.

Virtually all new instruction sets since 1982 have followed this RISC philosophy of fixed instruction lengths, load-store instruction sets, limited addressing modes, and limited operations. MIPS, Sun SPARC, Hewlett-Packard PA-RISC, IBM PowerPC, and DEC Alpha are all examples of RISC architectures.

A Brief History of the 80x86

The ancestors of the 80x86 were the first microprocessors, produced late in the first half of the 1970s. The Intel 4004 and 8008 were extremely simple 4-bit and 8-bit accumulator-style machines. Morse et al. [1980] describe the evolution of the 8086 from the 8080 in the late 1970s in an attempt to provide a 16-bit machine with better throughput. At that time, almost all programming for microprocessors was done in assembly language—both memory and compilers were in short supply. Intel wanted to keep its base of 8080 users, so the 8086 was designed to be "compatible" with the 8080. The 8086 was *never* object-code compatible with the 8080, but the machines were close enough that translation of assembly language programs could be done automatically.

In early 1980, IBM selected a version of the 8086 with an 8-bit external bus, called the 8088, for use in the IBM PC. They chose the 8-bit version to reduce the cost of the machine. This choice, together with the tremendous success of the IBM PC, has made the 8086 architecture ubiquitous. The success of the IBM

PC was due in part because IBM opened the architecture of the PC and enabled the PC-clone industry to flourish. As discussed in section 3.12, the 80286, 80386, 80486, Pentium, and Pentium Pro have extended the architecture and provided a series of performance enhancements.

Although the 68000 was chosen for the Macintosh, the Mac was never as pervasive as the PC, partly because Apple did not allow Mac clones based on the 68000, and the 68000 did not acquire the same software leverage that the 8086 enjoys. The Motorola 68000 may have been more significant *technically* than the 8086, but the impact of the selection by IBM and IBM's open architecture strategy dominated the technical advantages of the 68000 in the market.

Some argue that the inelegance of the 80x86 instruction set is unavoidable, the price that must be paid for rampant success by any architecture. We reject that notion. Obviously no successful architecture can jettison features that were added in previous implementations, and over time some features may be seen as undesirable. The awkwardness of the 80x86 begins at its core with the 8086 instruction set, and was exacerbated by the architecturally inconsistent expansions found in the 8087, 80286, 80386, and MMX.

A counterexample is the IBM 360/370 architecture, which is much older than the 80x86. It dominates the mainframe market just as the 80x86 dominates the PC market. Due undoubtedly to a better base and more compatible enhancements, this instruction set makes much more sense than the 80x86 more than 30 years after its first implementation.

Hewlett-Packard and Intel will announce a new, common instruction set architecture in about 1998. It will be upwards compatible with the 80x86, and thus the 80x86 instruction will be available in some form in computers of the next century.

Instruction set anthropologists of the 21st century will peel off layer after layer from such machines until they uncover artifacts from the first microprocessor. Given such a find, how will they judge 20th-century computer architecture?

To Probe Further

Bayko, J. [1996]. "Great Microprocessors of the Past and Present," available at *www.mkp.com/books_catalog/cod/links.htm*.

A personal view of the history of representative or unusual microprocessors, from the Intel 4004 to the Patriot Scientific ShBoom!

Kane, G., and J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ.

This book describes the MIPS architecture in greater detail than Appendix A.

Levy, H., and R. Eckhouse [1989]. *Computer Programming and Architecture: The VAX*, Digital Press, Boston.

This book concentrates on the VAX, but also includes descriptions of the Intel 80x86, IBM 360, and CDC 6600.

Morse, S., B. Ravenal, S. Mazor, and W. Pohlman [1980]. "Intel Microprocessors—8080 to 8086," *Computer* 13:10 (October).

The architecture history of the Intel from the 4004 to the 8086, according to the people who participated in the designs.

Wakerly, J. [1989]. *Microcomputer Architecture and Programming*, Wiley, New York.

The Motorola 680x0 is the main focus of the book, but it covers the Intel 8086, Motorola 6809, TI 9900, and Zilog Z8000.

3.16 Key Terms

The terms listed below reflect the key ideas discussed in this chapter. If you're unsure of the meaning of any of these terms, refer to the Glossary for a full definition.

activation record
address
addressing mode
base or displacement addressing
basic block
callee
caller
conditional branch
data transfer instruction
executable file
frame pointer
general-purpose register (GPR)
global pointer
immediate addressing
instruction format
instruction set
jump address table
jump-and-link instruction
linker or link editor
load-store or register-register machine
loader
object program
opcode
PC-relative addressing
procedure
procedure frame
program counter (PC)
pseudoinstruction
register addressing
return address
stack
stack pointer
stored-program computer
stored-program concept
word

3.17 Exercises

Appendix A describes the MIPS simulator, which is helpful for these exercises. Although the simulator accepts pseudoinstructions, try not to use pseudoinstructions for any exercises that ask you to produce MIPS code. Your goal should be to learn the real MIPS instruction set, and if you are asked to count instructions, your count should reflect the actual instructions that will be executed and not the pseudoinstructions.

There are some cases where pseudoinstructions must be used (for example, the `la` instruction when an actual value is not known at assembly time). In many cases they are quite convenient and result in more readable code (for example, the `li` and `move` instructions). If you choose to use pseudoinstructions for these reasons, please add a sentence or two to your solution stating which pseudoinstructions you have used and why.

3.1 [5] <§§3.3, 3.5, 3.8> Add comments to the following MIPS code and describe in one sentence what it computes. Assume that `$a0` is used for the input and initially contains `n`, a positive integer. Assume that `$v0` is used for the output.

```
begin:  addi $t0, $zero, 0
        addi $t1, $zero, 1
loop:   slt  $t2, $a0, $t1
        bne  $t2, $zero, finish
        add  $t0, $t0, $t1
        addi $t1, $t1, 2
        j    loop
finish: add  $v0, $t0, $zero
```

3.2 [12] <§§3.3, 3.5, 3.8> The following code fragment processes an array and produces two important values in registers `$v0` and `$v1`. Assume that the array consists of 5000 words indexed 0 through 4999, and its base address is stored in `$a0` and its size (5000) in `$a1`. Describe in one sentence what this code does. Specifically, what will be returned in `$v0` and `$v1`?

```
        add   $a1, $a1, $a1
        add   $a1, $a1, $a1
        add   $v0, $zero, $zero
        add   $t0, $zero, $zero
outer:  add   $t4, $a0, $t0
        lw    $t4, 0($t4)
        add   $t5, $zero, $zero
        add   $t1, $zero, $zero
inner:  add   $t3, $a0, $t1
        lw    $t3, 0($t3)
        bne   $t3, $t4, skip
        addi  $t5, $t5, 1
skip:   addi  $t1, $t1, 4
        bne   $t1, $a1, inner
        slt   $t2, $t5, $v0
        bne   $t2, $zero, next
        add   $v0, $t5, $zero
        add   $v1, $t4, $zero
next:   addi  $t0, $t0, 4
        bne   $t0, $a1, outer
```

3.3 [10] <§§3.3, 3.5, 3.8> Assume that the code from Exercise 3.2 is run on a machine with a 500-MHz clock that requires the following number of cycles for each instruction:

Instruction	Cycles
add,addi,slt	1
lw, bne	2

In the worst case, how many seconds will it take to execute this code?

3.4 [5] <§3.8> Show the single MIPS instruction or minimal sequence of instructions for this C statement:

```
a = b + 100;
```

Assume that `a` corresponds to register `$t0` and `b` corresponds to register `$t1`.

3.5 [10] <§3.8> Show the single MIPS instruction or minimal sequence of instructions for this C statement:

```
x[10] = x[11] + c;
```

Assume that `c` corresponds to register `$t0` and the array `x` has a base address of $4{,}000{,}000_{ten}$.

3.6 [10] <§§ 3.3, 3.5, 3.8> The following program tries to copy words from the address in register `$a0` to the address in register `$a1`, counting the number of words copied in register `$v0`. The program stops copying when it finds a word equal to 0. You do not have to preserve the contents of registers `$v1`, `$a0`, and `$a1`. This terminating word should be copied but not counted.

```
loop: lw    $v1,0($a0)      # Read next word from source
      addi  $v0,$v0,1       # Increment count words copied
      sw    $v1,0($a1)      # Write to destination
      addi  $a0,$a0,1       # Advance pointer to next source
      addi  $a1,$a1,1       # Advance pointer to next dest
      bne   $v1,$zero,loop  # Loop if word copied ≠ zero
```

There are multiple bugs in this MIPS program; fix them and turn in a bug-free version. Like many of the exercises in this chapter, the easiest way to write MIPS programs is to use the simulator described in Appendix A. (Go to *www.mkp.com/cod2e.htm* to get a copy of this program.)

3.7 [15] <§3.4> Using the MIPS program in Exercise 3.6 (with bugs intact), determine the instruction format for each instruction and the decimal values of each instruction field.

3.8 [10] <§§3.2, 3.3, 3.5, 3.8> {Ex. 3.6} Starting with the corrected program in the answer to Exercise 3.6, write the C code segment that might have produced this code. Assume that variable `source` corresponds to register `$a0`, variable `destination` corresponds to register `$a1`, and variable `count` corresponds to register `$v0`. Show variable declarations, but assume that `source` and `destination` have been initialized to the proper addresses.

3.9 [10] <§3.5> The C segment

```
while (save[i] == k)
  i = i + j;
```

on page 127 uses both a conditional branch and an unconditional jump each time through the loop. Only poor compilers would produce code with this loop overhead. Rewrite the assembly code so that it uses at most one branch or jump each time through the loop. How many instructions are executed before and after the optimization if the number of iterations of the loop is 10 (i.e., `save[i + 10 * j]` do not equal `k` and `save[i]`, . . . , `save[i + 9 * j]` equal `k`)?

3.10 [25] <§3.9> As discussed on page 157 and summarized in Figure 3.37, pseudoinstructions are not part of the MIPS instruction set but often appear in MIPS programs. For each pseudoinstruction in the following table, produce a minimal sequence of actual MIPS instructions to accomplish the same thing. You may need to use `$at` for some of the sequences. In the following table, `big` refers to a specific number that requires 32 bits to represent and `small` to a number that can be expressed using 16 bits.

Pseudoinstruction	What it accomplishes
move $t5, $t3	$t5 = $t3
clear $t5	$t5 = 0
li $t5, small	$t5 = small
li $t5, big	$t5 = big
lw $t5, big($t3)	$t5 = Memory[$t3 + big]
addi $t5, $t3, big	$t5 = $t3 + big
beq $t5, small, L	if ($t5 = small) go to L
beq $t5, big, L	if ($t5 = big) go to L
ble $t5, $t3, L	if ($t5 <= $t3) go to L
bgt $t5, $t3, L	if ($t5 > $t3) go to L
bge $t5, $t3, L	if ($t5 >= $t3) go to L

3.11 [30] <§3.5> Consider the following fragment of C code:

```
for (i=0; i<=100; i=i+1) {a[i] = b[i] + c;}
```

Assume that `a` and `b` are arrays of words and the base address of `a` is in `$a0` and the base address of `b` is in `$a1`. Register `$t0` is associated with variable `i` and register `$s0` with `c`. Write the code for MIPS. How many instructions are executed during the running of this code? How many memory data references will be made during execution?

3.12 [5] <§§3.8, 3.9> Given your understanding of PC-relative addressing, explain why an assembler might have problems directly implementing the branch instruction in the following code sequence:

```
here:   beq $t1, $t2, there
. . .
there:  add $t1, $t1, $t1
```

Show how the assembler might rewrite this code sequence to solve these problems.

3.13 [10] <§3.12> Consider an architecture that is similar to MIPS except that it supports update addressing (like the PowerPC) for data transfer instructions. If we run gcc using this architecture, some percentage of the data transfer instructions shown in Figure 3.38 on page 189 will be able to make use of the new instructions, and for each instruction changed, one arithmetic instruction can be eliminated. If 25% of the data transfer instructions can be changed, which will be faster for gcc, the modified MIPS architecture or the unmodified architecture? How much faster? (You can assume that both architectures have CPI values as given in Exercise 3.16 and that the modified architecture has its cycle time increased by 10% in order to accommodate the new instructions.)

3.14 [10] <§3.14> When designing memory systems, it becomes useful to know the frequency of memory reads versus writes as well as the frequency of accesses for instructions versus data. Using the average instruction-mix information for MIPS for the program gcc in Figure 3.38 on page 189, find the following:

a. The percentage of *all* memory accesses that are for data (vs. instructions).

b. The percentage of *all* memory accesses that are reads (vs. writes). Assume that two-thirds of data transfers are loads.

3.15 [10] <§3.14> Perform the same calculations as for Exercise 3.14, but replace the program gcc with spice.

3.16 [15] <§3.14> Suppose we have made the following measurements of average CPI for instructions:

Instruction	Average CPI
Arithmetic	1.0 clock cycles
Data transfer	1.4 clock cycles
Conditional branch	1.7 clock cycles
Jump	1.2 clock cycles

Compute the effective CPI for MIPS. Average the instruction frequencies for gcc and spice in Figure 3.38 on page 189 to obtain the instruction mix.

3.17 [20] <§3.10> In this exercise, we'll examine quantitatively the pros and cons of adding an addressing mode to MIPS that allows arithmetic instructions to directly access memory, as is found on the 80x86. The primary benefit is that fewer instructions will be executed because we won't have to first load a register. The primary disadvantage is that the cycle time will have to increase to account for the additional time to read memory. Consider adding a new instruction:

```
addm $t2, 100($t3) # $t2 = $t2 + Memory[$t3+100]
```

Assume that the new instruction will cause the cycle time to increase by 10%. Use the instruction frequencies for the gcc benchmark from Figure 3.38 on page 189, and assume that two-thirds of the data transfers are loads and the rest are stores. Assume that the new instruction affects only the clock speed, not the CPI. What percentage of loads must be eliminated for the machine with the new instruction to have at least the same performance?

3.18 [10] <§3.10> Using the information in Exercise 3.17, write a multiple-instruction sequence in which a load of `$t0` followed immediately by the use of `$t0`—in, say, an add—could *not* be replaced by a single instruction of the form proposed.

In More Depth

Comparing Instruction Sets of Different Styles

For the next two exercises, your task is to compare the memory efficiency of four different styles of instruction sets for two code sequences. The architecture styles are the following:

- *Accumulator.*
- *Memory-memory*: All three operands of each instruction are in memory.

- *Stack*: All operations occur on top of the stack. Only push and pop access memory, and all other instructions remove their operands from the stack and replace them with the result. The implementation uses a stack for the top two entries; accesses that use other stack positions are memory references.
- *Load-store*: All operations occur in registers, and register-to-register instructions have three operands per instruction. There are 16 general-purpose registers, and register specifiers are 4 bits long.

Consider the following C code:

```
a = b + c;    # a, b, and c are variables in memory
```

Section 3.15 contains the equivalent assembly language code for the different styles of instruction sets. For a given code sequence, we can calculate the instruction bytes fetched and the memory data bytes transferred using the following assumptions about all four instruction sets:

- The opcode is always 1 byte (8 bits).
- All memory addresses are 2 bytes (16 bits).
- All data operands are 4 bytes (32 bits).
- All instructions are an integral number of bytes in length.
- There are no optimizations to reduce memory traffic.

For example, a register load will require four instruction bytes (one for the opcode, one for the register destination, and two for a memory address) to be fetched from memory along with four data bytes. A memory-memory add instruction will require seven instruction bytes (one for the opcode and two for each of the three memory addresses) to be fetched from memory and will result in 12 data bytes being transferred (eight from memory to the processor and four from the processor back to memory). The following table displays a summary of this information for each of the architectural styles for the code appearing above and in section 3.15:

Style	Instructions for a = b + c	Code bytes	Data bytes
Accumulator	3	3 + 3 + 3	4 + 4 + 4
Memory-memory	1	7	12
Stack	4	3 + 3 + 1 + 3	4 + 4 + 0 + 4
Load-store	4	4 + 4 + 3 + 4	4 + 4 + 0 + 4

3.19 [20] <§3.15> For the following C code, write an equivalent assembly language program in each architectural style (assume all variables are initially in memory):

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

For each code sequence, calculate the instruction bytes fetched and the memory data bytes transferred (read or written). Which architecture is most efficient as measured by code size? Which architecture is most efficient as measured by total memory bandwidth required (code + data)? If the answers are not the same, why are they different?

3.20 [5] <§3.15> Sometimes architectures are characterized according to the typical number of memory addresses per instruction. Commonly used terms are 0, 1, 2, and 3 addresses per instruction. Associate the names above with each category.

3.21 [10] <§3.7> Compute the decimal byte values that form the null-terminated ASCII representation of the following string:

```
A byte is 8 bits
```

3.22 [30] <§§3.6, 3.7> Write a program in MIPS assembly language to convert an ASCII decimal string to an integer. Your program should expect register `$a0` to hold the address of a null-terminated string containing some combination of the digits 0 through 9. Your program should compute the integer value equivalent to this string of digits, then place the number in register `$v0`. Your program need not handle negative numbers. If a nondigit character appears anywhere in the string, your program should stop with the value –1 in register `$v0`. For example, if register `$a0` points to a sequence of three bytes 50_{ten}, 52_{ten}, 0_{ten} (the null-terminated string "24"), then when the program stops, register `$v0` should contain the value 24_{ten}. (The subscript "ten" means base 10.)

3.23 [20] <§§3.6, 3.7> Write a procedure, `bfind`, in MIPS assembly language. The procedure should take a single argument that is a pointer to a null-terminated string in register `$a0`. The `bfind` procedure should locate the first `b` character in the string and return its address in register `$v0`. If there are no `b`'s in the string, then `bfind` should return a pointer to the null character at the end of the string. For example, if the argument to `bfind` points to the string "`imbibe`," then the return value will be a pointer to the third character of the string.

3.24 [20] <§§3.6, 3.7> {Ex. 3.23} Write a procedure, `bcount`, in MIPS assembly language. The `bcount` procedure takes a single argument, which is a pointer to a string in register `$a0`, and it returns a count of the total number of `b` characters in the string in register `$v0`. You must use your `bfind` procedure in Exercise 3.23 in your implementation of `bcount`.

3.25 [30] <§§3.6, 3.7> Write a procedure, `itoa`, in MIPS assembly language that will convert an integer argument into an ASCII decimal string. The procedure should take two arguments: the first is an integer in register `$a0`; the second is the address at which to write a result string in register `$a1`. Then `itoa` should convert its first argument to a null-terminated decimal ASCII string and store that string at the given result location. The return value from `itoa`, in register `$v0`, should be a count of the number of non-null characters stored at the destination.

In More Depth

Tail Recursion

Some recursive procedures can be implemented iteratively without using recursion. Iteration can significantly improve performance by removing the overhead associated with procedure calls. For example, consider a procedure used to accumulate a sum:

```
int sum (int n, int acc) {
  if (n > 0)
      return sum(n - 1, acc + n);
  else
      return acc;
}
```

Consider the procedure call `sum(3,0)`. This will result in recursive calls to `sum(2,3)`, `sum(1,5)`, and `sum(0,6)`, and then the result 6 will be returned four times. This recursive call of sum is referred to as a *tail call*, and this example use of tail recursion can be implemented very efficiently (assume `$a0 = n` and `$a1 = acc`):

```
sum:    beq  $a0, $zero, sum_exit # go to sum_exit if n is 0
        add  $a1, $a1, $a0        # add n to acc
        addi $a0, $a0, -1         # subtract 1 from n
        j sum                     # go to sum
sum_exit:
        move $v0, $a1             # return value acc
        jr   $ra                  # return to caller
```


3.26 [30] <§3.6> Write a MIPS procedure to compute the nth Fibonacci number $F(n)$ where

$$F(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{otherwise.} \end{cases}$$

Base your algorithm on the straightforward but hopelessly inefficient procedure below, which generates a recursive process:

```
int fib(int n){
  if (n == 0)
      return 0;
  else if (n == 1)
      return 1;
  else
return fib(n-1) + fib(n-2);
```

3.27 [30] <§3.6> Write a program as in Exercise 3.26, except this time base your program on the following procedure and optimize the tail call so as to make your implementation efficient:

```
int fib_iter (int a, int b, int count) {
  if (count == 0)
      return b;
  else
      return fib_iter(a + b, a, count - 1);
```

Here, the first two parameters keep track of the previous two Fibonacci numbers computed. To compute `F(n)` you have to make the procedure call `fib_iter(1, 0, n)`.

3.28 [20] <§3.6> Estimate the difference in performance between your solution to Exercise 3.26 and your solution to Exercise 3.27.

In More Depth

The Single Instruction Computer

The computer architecture used in this book, MIPS, has one of the simpler instruction sets in existence. However, it is possible to imagine even simpler instruction sets. In this assignment, you are to consider a hypothetical machine called SIC, for Single Instruction Computer. As its name implies, SIC has only one instruction: subtract and branch if negative, or `sbn` for short. The `sbn` instruction has three operands, each consisting of the address of a word in memory:

```
sbn a,b,c # Mem[a] = Mem[a] - Mem[b];if (Mem[a]<0) go to c
```

The instruction will subtract the number in memory location `b` from the number in location `a` and place the result back in `a`, overwriting the previous value. If the result is greater than or equal to 0, the computer will take its next instruction from the memory location just after the current instruction. If the result is less than 0, the next instruction is taken from memory location `c`. SIC has no registers and no instructions other than `sbn`.

Although it has only one instruction, SIC can imitate many of the operations of more complex instruction sets by using clever sequences of `sbn` instructions. For example, here is a program to copy a number from location `a` to location `b`:

```
start:  sbn temp,temp,.+1   # Sets temp to zero
        sbn temp,a,.+1      # Sets temp to -a
        sbn b,b,.+1         # Sets b to zero
        sbn b,temp,.+1      # Sets b to -temp, which is a
```

In the program above, the notation `.+1` means "the address after this one," so that each instruction in this program goes on to the next in sequence whether or not the result is negative. We assume `temp` to be the address of a spare memory word that can be used for temporary results.

3.29 [10] <§3.15> Write a SIC program to add `a` and `b`, leaving the result in `a` and leaving `b` unmodified.

3.30 [20] <§3.15> Write a SIC program to multiply `a` by `b`, putting the result in `c`. Assume that memory location `one` contains the number 1. Assume that `a` and `b` are greater than 0 and that it's OK to modify `a` or `b`. (Hint: What does this program compute?)

```
c = 0; while (b > 0) {b = b - 1; c = c + a;}
```

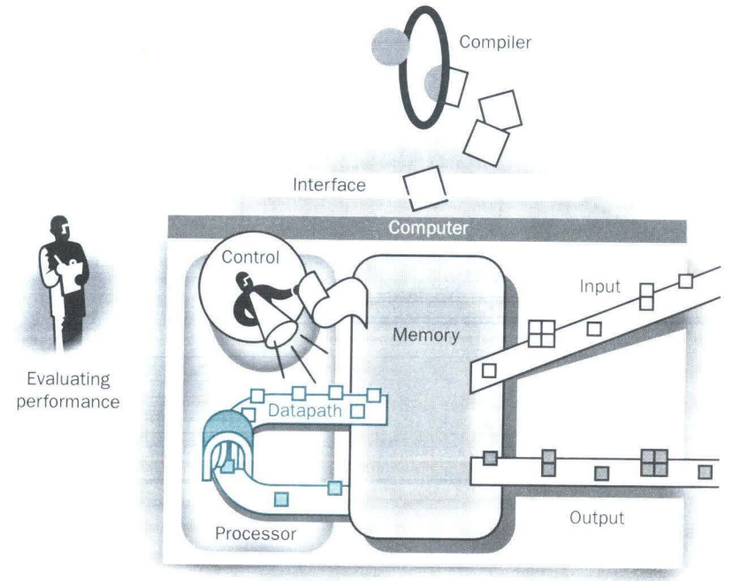
4

Arithmetic for Computers

Numerical precision is the very soul of science.

Sir D'arcy Wentworth Thompson
On Growth and Form, 1917

The Five Classic Components of a Computer



4.1 Introduction

Computer words are composed of bits; thus words can be represented as binary numbers. Although the natural numbers 0, 1, 2, and so on can be represented either in decimal or binary form, what about the other numbers that commonly occur? For example:

- How are negative numbers represented?
- What is the largest number that can be represented in a computer word?
- What happens if an operation creates a number bigger than can be represented?
- What about fractions and real numbers?

We could also ask, What is the inside story about the infamous bug in the Pentium? And underlying all these questions is a mystery: How does hardware really add, subtract, multiply, or divide numbers?

The goal of this chapter is to unravel this mystery, including representation of numbers, arithmetic algorithms, hardware that follows these algorithms, and the implications of all this for instruction sets. These insights may even explain quirks that you have already encountered with computers. (If you are familiar with signed binary numbers, you may wish to skip the next section and go to section 4.3 on page 220.)

4.2 Signed and Unsigned Numbers

Numbers can be represented in any base; humans prefer base 10 and, as we examined in Chapter 3, base 2 is best for computers. Because we will frequently be dealing with both decimal and binary numbers, to avoid confusion we will subscript decimal numbers with *ten* and binary numbers with *two*.

In any number base, the value of ith digit d is

$$d \times \text{Base}^i$$

where i starts at 0 and increases from right to left. This leads to an obvious way to number the bits in the word: Simply use the power of the base for that bit. For example,

1011_{two}

represents

$$\begin{aligned}&(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{ten}\\ &= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{ten}\\ &= 8 + 0 + 2 + 1_{ten}\\ &= 11_{ten}\end{aligned}$$

Hence the bits are numbered 0, 1, 2, 3, . . . from *right to left* in a word. The drawing below shows the numbering of bits within a MIPS word and the placement of the number 1011_{two}:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

Since words are drawn vertically as well as horizontally, leftmost and rightmost may be unclear. Hence, the phrase *least significant bit* is used to refer to the rightmost bit (bit 0 above) and *most significant bit* to the leftmost bit (bit 31).

The MIPS word is 32 bits long, so we can represent 2^{32} different 32-bit patterns. It is natural to let these combinations represent the numbers from 0 to $2^{32} - 1$ ($4{,}294{,}967{,}295_{ten}$):

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
. . . . . .
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$

Hardware Software Interface

Base 2 is not natural to human beings; we have 10 fingers and so find base 10 natural. Why didn't computers use decimal? In fact, the first commercial computer *did* offer decimal arithmetic. The problem was that the computer still used on and off signals, so a decimal digit was simply represented by several binary digits. Decimal proved so inefficient that subsequent machines reverted to all binary, converting to base 10 only for the infrequent input/output events.

ASCII versus Binary Numbers

Example We could represent numbers as strings of ASCII digits instead of as two's complement integers (see Figure 3.15 on page 142). What is the expansion in storage if the number 1 billion is represented in ASCII versus a 32-bit integer?

Answer One billion is 1 000 000 000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be $(10 \times 8)/32$ or 2.5. In addition to the expansion in storage, the hardware to add, subtract, multiply, and divide such numbers is also difficult. Such difficulties explain why computing professionals are raised to believe that binary is natural and that the occasional decimal machine is bizarre.

Keep in mind that the binary bit patterns above are simply *representatives* of numbers. Numbers really have an infinite number of digits, with almost all being 0 except for a few of the rightmost digits. We just don't normally show leading 0s.

As we shall see in sections 4.5 through 4.7, hardware can be designed to add, subtract, multiply, and divide these binary bit patterns. If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, *overflow* is said to have occurred. It's up to the operating system and program to determine what to do if overflow occurs.

Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative. The most obvious solution is to add a separate sign, which conveniently can be represented in a single bit; the name for this representation is *sign and magnitude*.

Alas, sign and magnitude representation has several shortcomings. First, it's not obvious where to put the sign bit. To the right? To the left? Early machines tried both. Second, adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be. Finally, a separate sign bit means that sign and magnitude has both a positive and negative zero, which can lead to problems for inattentive programmers. As a result of these shortcomings, sign and magnitude was soon abandoned.

In the search for a more attractive alternative, the question arose as to what would be the result for unsigned numbers if we tried to subtract a large number from a small one. The answer is that it would try to borrow from a string of leading 0s, so the result would have a string of leading 1s.

Given that there was no obvious better alternative, the final solution was to pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative. This convention for representing signed binary numbers is called *two's complement* representation:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
. . . . . .

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
. . . . . .

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

The positive half of the numbers, from 0 to $2{,}147{,}483{,}647_{ten}$ ($2^{31}-1$), use the same representation as before. The following bit pattern ($1000\ldots0000_{two}$) represents the most negative number $-2{,}147{,}483{,}648_{ten}$ (-2^{31}). It is followed by a declining set of negative numbers: $-2{,}147{,}483{,}647_{ten}$ ($1000\ldots0001_{two}$) down to -1_{ten} ($1111\ldots1111_{two}$).

Two's complement does have one negative number, $-2{,}147{,}483{,}648_{ten}$, that has no corresponding positive number. Such imbalance was a worry to the inattentive programmer, but sign and magnitude had problems for both the programmer *and* the hardware designer. Consequently, every computer today uses two's complement binary representations for signed numbers.

Two's complement representation has the advantage that all negative numbers have a 1 in the most significant bit. Consequently, hardware needs to test only this bit to see if a number is positive or negative (with 0 considered positive). This particular bit is often called the *sign bit*. By recognizing the role of the sign bit, we can represent positive and negative numbers in terms of the bit value times a power of 2 (here xi means the ith bit of x):

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

The sign bit is multiplied by -2^{31}, and the rest of the bits are then multiplied by positive versions of their respective base values.

Binary to Decimal Conversion

Example What is the decimal value of this 32-bit two's complement number?

1111 1111 1111 1111 1111 1111 1111 1100_{two}

Answer Substituting the number's bit values into the formula above:

$(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0})$
$= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0$
$= -2{,}147{,}483{,}648_{ten} + 2{,}147{,}483{,}644_{ten}$
$= -4_{ten}$

We'll see a shortcut to simplify conversion soon.

Hardware Software Interface

Signed versus unsigned applies to loads as well as to arithmetic. The *function* of a signed load is to copy the sign repeatedly to fill the rest of the register—called *sign extension*—but its *purpose* is to place a correct representation of the number within that register. Unsigned loads simply fill with 0s to the left of the data, since the number represented by the bit pattern is unsigned.

When loading a 32-bit word into a 32-bit register, the point is moot; signed and unsigned loads are identical. MIPS does offer two flavors of byte loads: *load byte* (`lb`) treats the byte as a signed number and thus sign extends to fill the 24 leftmost bits of the register, while *load byte unsigned* (`lbu`) works with unsigned integers. Since programs almost always use bytes to represent characters rather than consider bytes as short signed integers, `lbu` is used practically exclusively for byte loads.

Just as an operation on unsigned numbers can overflow the capacity of hardware to represent the result, so can an operation on two's complement numbers. Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect): a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive.

Hardware Software Interface

Unlike the numbers discussed above, memory addresses naturally start at 0 and continue to the largest address. Put another way, negative addresses make no sense. Thus, programs want to deal sometimes with numbers that can be positive or negative and sometimes with numbers that can be only positive. Programming languages reflect this distinction. C, for example, names the former *integers* (declared as `int` in the program) and the latter *unsigned integers* (`unsigned int`).

Comparison instructions must deal with this dichotomy. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0.

MIPS offers two versions of the set on less than comparison to handle these alternatives. *Set on less than* (`slt`) and *set on less than immediate* (`slti`) work with signed integers. Unsigned integers are compared using *set on less than unsigned* (`sltu`) and *set on less than immediate unsigned* (`sltiu`).

Signed versus Unsigned Comparison

Example Suppose register `$s0` has the binary number

1111 1111 1111 1111 1111 1111 1111 1111_{two}

and that register `$s1` has the binary number

0000 0000 0000 0000 0000 0000 0000 0001_{two}

What are the values of registers `$t0` and `$t1` after these two instructions?

```
slt     $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

Answer The value in register `$s0` represents –1 if it is an integer and $4{,}294{,}967{,}295_{ten}$ if it is an unsigned integer. The value in register `$s1` represents 1 in either case. Then register `$t0` has the value 1, since $-1_{ten} < 1_{ten}$, and register `$t1` has the value 0, since $4{,}294{,}967{,}295_{ten} > 1_{ten}$.

Before going on to addition and subtraction, let's examine a few useful shortcuts when working with two's complement numbers.

The first shortcut is a quick way to negate a two's complement binary number. Simply invert every 0 to 1 and every 1 to 0, then add one to the result. This shortcut is based on the observation that the sum of a number and its inverted representation must be $111\ldots111_{two}$, which represents –1. Since $x + \bar{x} \equiv -1$, therefore $x + \bar{x} + 1 = 0$ or $\bar{x} + 1 = -x$.

Negation Shortcut

Example Negate 2_{ten}, and then check the result by negating -2_{ten}.

Answer

$$2_{ten} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$$

Negating this number by inverting the bits and adding one,

$$\begin{array}{rl} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} \\ = & -2_{ten} \end{array}$$

Going the other direction,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

is first inverted and then incremented:

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} \\ = & 2_{ten} \end{array}$$

The second shortcut tells us how to convert a binary number represented in n bits to a number represented with more than n bits. For example, the immediate field in the load, store, branch, add, and set on less than instructions contains a two's complement 16-bit number, representing $-32{,}768_{ten}$ (-2^{15}) to $32{,}767_{ten}$(2^{15}–1). To add the immediate field to a 32-bit register, the machine must convert that 16-bit number to its 32-bit equivalent. The shortcut is to take the most significant bit from the smaller quantity—the sign bit—and replicate it to fill the new bits of the larger quantity. The old bits are simply copied into the right portion of the new word. This shortcut is commonly called *sign extension*.

Sign Extension Shortcut

Example Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 32-bit binary numbers.

Answer The 16-bit binary version of the number 2 is

$$0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

It is converted to a 32-bit number by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word. The right half gets the old value:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

Let's negate the 16-bit version of 2 using the earlier shortcut. Thus,

$$0000\ 0000\ 0000\ 0010_{two}$$

becomes

$$\begin{array}{rl} & 1111\ 1111\ 1111\ 1101_{two} \\ + & \qquad\qquad\qquad 1_{two} \\ \hline = & 1111\ 1111\ 1111\ 1110_{two} \end{array}$$

Creating a 32-bit version of the negative number means copying the sign bit 16 times and placing it on the left:

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$$

This trick works because positive two's complement numbers really have an infinite number of 0s on the left and those that are negative two's complement numbers have an infinite number of 1s. The binary bit pattern representing a number hides leading bits to fit the width of the hardware; sign extension simply restores some of them.

A final shortcut, which we previewed in Chapter 3, is that we can save reading and writing long binary numbers by using a higher base than binary that converts easily into binary. Since almost all computer data sizes are multiples of 4, *hexadecimal* (base 16) numbers are popular. Since base 16 is a power of 2, we can trivially convert by replacing each group of four binary digits by a single hexadecimal digit, and vice versa. Figure 4.1 shows the hexadecimal Rosetta stone. We will use either the subscript *hex* or the C notation, which uses 0x*nnnn*, for hexadecimal numbers.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

FIGURE 4.1 The hexadecimal-binary conversion table. Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of four, go from right to left.

Binary-to-Hexadecimal Shortcut

Example

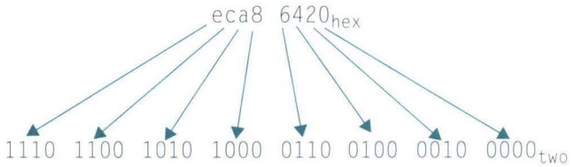
Convert the following hexadecimal and binary numbers into the other base:

$$eca8\ 6420_{hex}$$

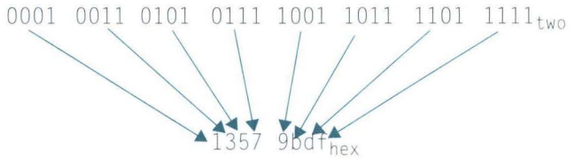
$$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_{two}$$

Answer

Just a table lookup one way:



And then the other direction:



Summary

The main point of this section is that we need to represent both positive and negative integers within a computer word, and although there are pros and cons to any option, the overwhelming choice since 1965 has been two's complement. Figure 4.2 shows the additions to the MIPS assembly language revealed in this section. (The MIPS machine language is also illustrated on the back endpapers of this book.)

MIPS operands

Name	Example	Comments
32 registers	`$s0–$s7, $t0–$t9, $gp, $fp, $zero, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. Register `$at` is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three operands
	add immediate	`addi $s1,$s2,100`	$s1 = $s2 + 100	+ constant
Data transfer	load word	`lw $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1,100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte unsigned	`lbu $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1,100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1,100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	`slti $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; two's complement
	set less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; unsigned numbers
	set less than immediate unsigned	`sltiu $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; unsigned numbers
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

FIGURE 4.2 MIPS architecture revealed thus far. Color indicates portions from this section added to the MIPS architecture revealed in Chapter 3 (Figure 3.20 on page 155). MIPS machine language is listed in the back endpapers of this book.

Elaboration: Two's complement gets its name from the rule that the unsigned sum of an n-bit number and its negative is 2^n, hence the complement or negation of a two's complement number x is $2^n - x$.

A third alternative representation is called *one's complement*. The negative of a one's complement is found by inverting each bit, from 0 to 1 and from 1 to 0, which helps explain its name since the complement of x is $2^n - x - 1$. It was also an attempt

to be a better solution than sign and magnitude, and several scientific computers did use the notation. This representation is similar to two's complement except that it also has two 0s: $00\ldots00_{two}$ is positive 0 and $11\ldots11_{two}$ is negative 0. The most negative number $10\ldots000_{two}$ represents $-2{,}147{,}483{,}647_{ten}$, and so the positives and negatives are balanced. One's complement adders did need an extra step to subtract a number, and hence two's complement dominates today.

A final notation, which we will look at when we discuss floating point, is to represent the most negative value by $00\ldots000_{two}$ and the most positive value represented by $11\ldots11_{two}$, with 0 typically having the value $10\ldots00_{two}$. This is called a *biased* notation, for it biases the number such that the number plus the bias has a non-negative representation.

4.3 Addition and Subtraction

Subtraction: Addition's Tricky Pal

> No. 10, Top Ten Courses for Athletes at a Football Factory,
> David Letterman et al., *Book of Top Ten Lists*, 1990

Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: The appropriate operand is simply negated before being added.

Binary Addition and Subtraction

Example

Let's try adding 6_{ten} to 7_{ten} in binary and then subtracting 6_{ten} from 7_{ten} in binary.

Answer

$$\begin{array}{rll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} & = 7_{ten} \\ + & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} & = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} & = 13_{ten} \end{array}$$

The 4 bits to the right have all the action; Figure 4.3 shows the sums and carries. The carries are shown in parentheses, with the arrows showing how they are passed.

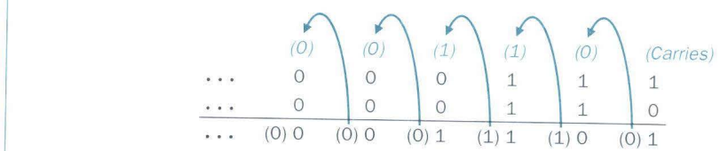


FIGURE 4.3 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$\begin{array}{rll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} & = 7_{ten} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} & = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} & = 1_{ten} \end{array}$$

or via addition using the two's complement representation of –6:

$$\begin{array}{rll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} & = 7_{ten} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} & = -6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} & = 1_{ten} \end{array}$$

We said earlier that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word. When can overflow occur in addition? When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, –10 + 4 = –6. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore no overflow can occur when adding positive and negative operands.

There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: When the signs of the operands are the *same*, overflow cannot occur. To see this, remember that $x - y = x + (-y)$ because we subtract by negating the second operand and then add. So, when we subtract operands of the same sign we end up by *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot occur in this case either.

Having examined when overflow cannot occur in addition and subtraction, we still haven't answered how to detect when it does occur. Overflow occurs when adding two positive numbers and the sum is negative, or vice versa. Clearly, adding or subtracting two 32-bit numbers can yield a result that needs

33 bits to be fully expressed. The lack of a 33rd bit means that when overflow occurs the sign bit is being set with the *value* of the result instead of the proper sign of the result. Since we need just one extra bit, only the sign bit can be wrong. This means a carry out occurred into the sign bit.

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. This means a borrow occurred from the sign bit. Figure 4.4 shows the combination of operations, operands, and results that indicate an overflow. (Exercise 4.42 gives a shortcut for detecting overflow more simply in hardware.)

We have just seen how to detect overflow for two's complement numbers in a machine. What about unsigned integers? Unsigned integers are commonly used for memory addresses where overflows are ignored.

The machine designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

- Add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow.
- Add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do *not* cause exceptions on overflow.

Because C ignores overflows, the MIPS C compilers will always generate the unsigned versions of the arithmetic instructions `addu`, `addiu`, and `subu` no matter what the type of the variables. The MIPS Fortran compilers, however, pick the appropriate arithmetic instructions, depending on the type of the operands.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 4.4 Overflow conditions for addition and subtraction.

Hardware Software Interface

The machine designer must decide how to handle arithmetic overflows. Although some languages like C leave the decision up to the machine designer, languages like Ada and Fortran require that the program be notified. The programmer or the programming environment must then decide what to do when overflow occurs.

MIPS detects overflow with an *exception*, also called an *interrupt* on many computers. An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed. (Section 5.6 covers exceptions in more detail; Chapters 7 and 8 describe other situations where exceptions and interrupts occur.)

MIPS includes a register called the *exception program counter* (*EPC*) to contain the address of the instruction that caused the exception. The instruction *move from system control* (`mfc0`) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the offending instruction via a jump register instruction.

Summary

The main point of this section is that, independent of the representation, the finite word size of computers means that arithmetic operations can create results that are too large to fit in this fixed word size. It's easy to detect overflow in unsigned numbers, although these are almost always ignored because programs don't want to detect overflow for address arithmetic, the most common use of natural numbers. Two's complement presents a greater challenge, yet some software systems require detection of overflow, so today all machines have a way to detect it. Figure 4.5 shows the additions to the MIPS architecture from this section.

Elaboration: MIPS can trap on overflow, but unlike many other machines there is no conditional branch to test overflow. A sequence of MIPS instructions can discover overflow. For signed addition, the sequence is the following (see the In More Depth section on page 329 for the definition of the `xor` and `nor` instructions):

```
addu $t0, $t1,    $t2          # $t0 = sum, but don't trap
xor  $t3, $t1,    $t2          # Check if signs differ
slt  $t3, $t3,    $zero        # $t3 = 1 if signs differ
bne  $t3, $zero,  No_overflow  # $t1, $t2 signs ≠, so no overflow
xor  $t3, $t0,    $t1          # signs =; sign of sum match too?
                               # $t3 negative if sum sign different
slt  $t3, $t3,    $zero        # $t3 = 1 if sum sign different
bne  $t3, $zero,  Overflow     # All three signs ≠: go to overflow
```

MIPS operands

Name	Example	Comments
32 registers	$s0–$s7, $t0–$t9, $gp, $fp, $zero, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to, for example, handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0 $s1,$epc	$s1 = $epc	Used to copy Exception PC plus other special registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte unsigned	lbu $s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; two's complement
	set less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; unsigned numbers
	set less than immediate unsigned	sltiu $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; unsigned numbers
Uncondi-tional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

FIGURE 4.5 MIPS architecture revealed thus far. Color indicates the portions revealed since Figure 4.2 on page 219. MIPS machine language is also listed on the back endpapers of this book.

For unsigned addition ($t0 = $t1 + $t2), the test is

```
addu $t0, $t1, $t2       # $t0 = sum
nor  $t3, $t1, $zero     # $t3 = NOT $t1
                         # (2's comp - 1: 2^32 - $t1 - 1)
sltu $t3, $t3, $t2       # (2^32 - $t1 - 1) < $t2
                         # ⇒ 2^32 - 1 < $t1 + $t2
bne  $t3,$zero, Overflow # if (2^32 - 1 < $t1 + $t2) go to overflow
```

Elaboration: In the preceding text, we said that you copy EPC into a register via mfc0 and then return to the interrupted code via jump register. This leads to an interesting question: Since you must first transfer EPC to a register to use with jump register, how can jump register return to the interrupted code *and* restore the original values of *all* registers? You either restore the old registers first, thereby destroying your return address from EPC that you placed in a register for use in jump register, or you restore all registers but the one with the return address so that you can jump—meaning an exception would result in changing that one register at any time during program execution! Neither option is satisfactory.

To rescue the hardware from this dilemma, MIPS programmers agreed to reserve registers $k0 and $k1 for the operating system; these registers are *not* restored on exceptions. Just as the MIPS compilers avoid using register $at so that the assembler can use it as a temporary register (see the Hardware Software Interface section on page 147 in Chapter 3), compilers also abstain from using registers $k0 and $k1 to make them available for the operating system. Exception routines place the return address in one of these registers and then use jump register to restore the instruction address.

4.4 Logical Operations

"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

Lewis Carroll, *Alice's Adventures in Wonderland*, 1865

Although the first computers concentrated on full words, it soon became clear that it was useful to operate on fields of bits within a word or even on individual bits. Examining characters within a word, each of which are stored as 8 bits, is one example of such an operation. It follows that instructions were added to simplify, among other things, the packing and unpacking of bits into words.

One class of such operations is called *shifts*. They move all the bits in a word to the left or right, filling the emptied bits with 0s. For example, if register `$s0` contained

0000 0000 0000 0000 0000 0000 0000 1101_{two}

and the instruction to shift left by eight was executed, the new value would look like this:

0000 0000 0000 0000 0000 0000 1101 0000_{two}

The dual of a shift left is a shift right. The actual name of the two MIPS shift instructions are called *shift left logical* (`sll`) and *shift right logical* (`srl`). The following instruction performs the operation above, assuming that the result should go in register `$t2`:

```
sll    $t2,$s0,8  # reg $t2 = reg $s0 << 8 bits
```

We delayed explaining the *shamt* field in the R-format in Chapter 3. It stands for *shift amount* and is used in shift instructions. Hence, the machine language version of the instruction above is

op	rs	rt	rd	shamt	funct
0	0	16	10	8	0

The encoding of `sll` is 0 in both the op and funct fields, rd contains `$t2`, rt contains `$s0`, and shamt contains 8. The rs field is unused, and thus is set to 0.

Another useful operation that isolates fields is *AND*. (We capitalize the word to avoid confusion between the operation and the English conjunction.) AND is a bit-by-bit operation that leaves a 1 in the result only if both bits of the operands are 1. For example, if register `$t2` still contains

0000 0000 0000 0000 0000 1101 0000 0000_{two}

and register `$t1` contains

0000 0000 0000 0000 0011 1100 0000 0000_{two}

then, after executing the MIPS instruction

```
and $t0,$t1,$t2  # reg $t0 = reg $t1 & reg $t2
```

the value of register `$t0` would be

0000 0000 0000 0000 0000 1100 0000 0000_{two}

As you can see, AND can be used to apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a *mask*, since the mask "conceals" some bits.

To place a value into one of these seas of 0s, there is the dual to AND, called *OR*. It is a bit-by-bit operation that places a 1 in the result if *either* operand bit is a 1. To elaborate, if the registers `$t1` and `$t2` are unchanged from the preceding example, the result of the MIPS instruction

```
or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2
```

is this value in register `$t0`:

0000 0000 0000 0000 0011 1101 0000 0000_{two}

Figure 4.6 shows the logical C operations and the corresponding MIPS instructions. Constants are useful in logical operations as well as in arithmetic operations, so MIPS also provides the instructions *and immediate* (`andi`) and *or immediate* (`ori`). This section describes the logical operations AND, OR, and shift found in every computer today. The logical instructions are highlighted in Figure 4.7, which summarizes the MIPS instructions seen thus far.

Logical operations	C operators	MIPS instructions
Shift left	<<	`sll`
Shift right	>>	`srl`
Bit-by-bit AND	&	`and, andi`
Bit-by-bit OR	|	`or, ori`

FIGURE 4.6 Logical operations and their corresponding operations in C and MIPS.

Hardware Software Interface

C allows *bit fields* or *fields* to be defined within words, both allowing objects to be packed within a word *and* to match an externally enforced interface such as an I/O device. All fields must fit within a single word. Fields are unsigned integers that can be as short as 1 bit. C compilers insert and extract fields using logical instructions in MIPS: `and`, `or`, `sll`, and `srl`.

MIPS operands

Name	Example	Comments
32 registers	`$s0–$s7`, `$t0–$t9`, `$gp`, `$fp`, `$zero`, `$sp`, `$ra`, `$at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. Register `$at` is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; overflow detected
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands; overflow detected
	add immediate	`addi $s1,$s2,100`	`$s1 = $s2 + 100`	+ constant; overflow detected
	add unsigned	`addu $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; overflow undetected
	subtract unsigned	`subu $s1,$s2,$s3`	`$s1 = $s2 – $s3`	Three operands; overflow undetected
	add immediate unsigned	`addiu $s1,$s2,100`	`$s1 = $s2 + 100`	+ constant; overflow undetected
	move from coprocessor register	`mfc0 $s1,$epc`	`$s1 = $epc`	Used to copy Exception PC plus other special registers
Logical	and	`and $s1,$s2,$s3`	`$s1 = $s2 & $s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1 = $s2 | $s3`	Three reg. operands; bit-by-bit OR
	and immediate	`andi $s1,$s2,100`	`$s1 = $s2 & 100`	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,100`	`$s1 = $s2 | 100`	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1 = $s2 << 10`	Shift left by constant
	shift right logical	`srl $$s1,$s2,10`	`$s1 = $s2 >> 10`	Shift right by constant
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load byte unsigned	`lbu $s1,100($s2)`	`$s1 = Memory[$s2 +100]`	Byte from memory to register
	store byte	`sb $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1,100`	`$s1 = 100 * 2^16`	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than; two's complement
	set less than immediate	`slti $s1,$s2,100`	if (`$s2 < 100`) `$s1 = 1`; else `$s1 = 0`	Compare < constant; two's complement
	set less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than; natural numbers
	set less than immediate unsigned	`sltiu $s1,$s2,100`	if (`$s2 < 100`) `$s1 = 1`; else `$s1 = 0`	Compare < constant; natural numbers
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

FIGURE 4.7 MIPS architecture revealed thus far. Color indicates the portions introduced since Figure 4.5 on page 224. MIPS machine language is also listed on the back endpapers of this book.

C Bit Fields

Example

The following C code allocates three fields with a word labeled `receiver`: a 1-bit field named `ready`, a 1-bit field named `enable`, and an 8-bit field named `receivedByte`. It copies `receivedByte` into `data`, sets `ready` to 0, and sets `enable` to 1.

```
int data;
struct
{
     unsigned int ready:        1;
     unsigned int enable:       1;
     unsigned int receivedByte:8;
}receiver;
...
     data = receiver.receivedByte;
     receiver.ready = 0;
     receiver.enable = 1;
```

What is the compiled MIPS code? Assume `data` and `receiver` are allocated to `$s0` and `$s1`.

Answer

The fields look like this in a word (C typically right-aligns fields):

The first step is to isolate the 8-bit field (`receivedByte`) by first shifting it as far to the left as possible and then as far to the right as possible:

```
sll     $s0, $s1, 22 # move 8-bit field to left end
srl     $s0, $s0, 24 # move 8-bit field to right end
```

The third instruction clears the least significant bit with the mask $fffe_{hex}$ and the last instruction sets its neighbor bit to 1:

```
andi    $s1, $s1, fffe_hex # bit 0 set to 0
ori     $s1, $s1, 0002_hex # bit 1 set to 1
```

Elaboration: In the example this alternative sequence works as well:

```
srl   $s0, $s1, 2
andi  $s0, $s0, 0x00ff
```

The field is in the lower 16 bits of the word and we want 0s in the upper bits of the result of the `andi`. In general, a shift left of 32 – (n + m) followed by a shift right by 32 – n will isolate any n-bit field whose least significant bit is in bit m.

Since `addi` and `slti` are intended for signed numbers, it is not surprising that their immediate fields are sign-extended before use. Branch and data transfer address fields are sign-extended as well.

Perhaps it *is* surprising that `addiu` and `sltiu` also sign-extend their immediates, but they do. The `u` stands for unsigned, but in reality `addiu` is often used simply as an `add` instruction that cannot overflow, and hence we often want to add negative numbers. It's much harder to come up with an excuse for why `sltiu` sign extends its immediate field.

Since `andi` and `ori` normally work with unsigned integers, the immediates are treated as unsigned integers as well, meaning that they are expanded to 32 bits by padding with leading 0s instead of sign extension. Thus if the bit fields in the third line of the example above extended beyond the 16 least significant bits, the `andi` instruction would need a 32-bit constant to avoid clearing the upper portion of the fields.

The MIPS assembler creates 32-bit constants with the pair of instructions `lui` and `ori`; see Chapter 3, page 147 for an example of creating 32-bit constants using `lui` and `addi`.

4.5 Constructing an Arithmetic Logic Unit

*ALU n. [**A**rthritic **L**ogic **U**nit or (rare) **A**rithmetic **L**ogic **U**nit] A random-number generator supplied as standard with all computer systems.*

Stan Kelly-Bootle, *The Devil's DP Dictionary*, 1981

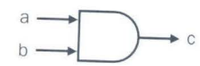
The *arithmetic logic unit* or *ALU* is the brawn of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR. This section constructs an ALU from the four hardware building blocks shown in Figure 4.8 (see Appendix B for more details on these building blocks). Cases 1, 2, and 4 in Figure 4.8 all have two inputs. We will sometimes use versions of these components with more than two inputs, confident that you can generalize from this simple example. In any case, Appendix B provides examples with more inputs. (You may wish to review sections B.1 through B.3 before proceeding further.)

Because the MIPS word is 32 bits wide, we need a 32-bit-wide ALU. Let's assume that we will connect 32 1-bit ALUs to create the desired ALU. We'll therefore start by constructing a 1-bit ALU.

A 1-Bit ALU

The logical operations are easiest, because they map directly onto the hardware components in Figure 4.8.

1. AND gate (c = a · b)



a	b	c = a · b
0	0	0
0	1	0
1	0	0
1	1	1

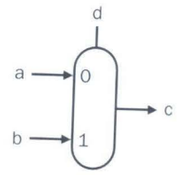
2. OR gate (c = a + b)

a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter (c = $\bar{a}$)

a	c = $\bar{a}$
0	1
1	0

4. Multiplexor (if d = = 0, c = a; else c = b)



d	c
0	a
1	b

FIGURE 4.8 Four hardware building blocks used to construct an arithmetic logic unit. The name of the operation and an equation describing it appear on the left. In the middle is the symbol for the block we will use in the drawings. On the right are tables that describe the outputs in terms of the inputs. Using the notation from Appendix B, a • b means "a AND b," a + b means "a OR b," and a line over the top (e.g., $\bar{a}$) means invert.

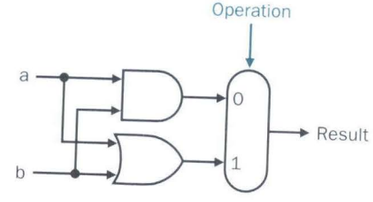


FIGURE 4.9 The 1-bit logical unit for AND and OR.

The 1-bit logical unit for AND and OR looks like Figure 4.9. The multiplexor on the right then selects *a* AND *b* or *a* OR *b*, depending on whether the value of *Operation* is 0 or 1. The line that controls the multiplexor is shown in color to distinguish it from the lines containing data. Notice that we have renamed the control and output lines of the multiplexor to give them names that reflect the function of the ALU.

The next function to include is addition. From Figure 4.3 on page 221 we can deduce the inputs and outputs of a single-bit adder. First, an adder must have two inputs for the operands and a single-bit output for the sum. There must be a second output to pass on the carry, called *CarryOut*. Since the CarryOut from the neighbor adder must be included as an input, we need a third input. This input is called *CarryIn*. Figure 4.10 shows the inputs and the outputs of a 1-bit adder. Since we know what addition is supposed to do, we can specify the outputs of this "black box" based on its inputs, as Figure 4.11 demonstrates.

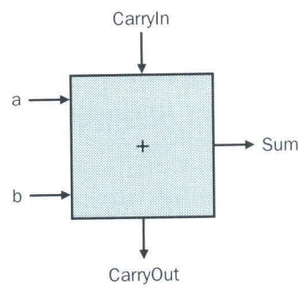


FIGURE 4.10 A 1-bit adder. This adder is called a full adder; it is also called a (3,2) adder because it has 3 inputs and 2 outputs. An adder with only the a and b inputs is called a (2,2) adder or half adder.

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

FIGURE 4.11 Input and output specification for a 1-bit adder.

From Appendix B, we know that we can express the output functions CarryOut and Sum as logical equations, and these equations can in turn be implemented with the building blocks in Figure 4.8. Let's do CarryOut. Figure 4.12 shows the values of the inputs when CarryOut is a 1.

We can turn this truth table into a logical equation, as explained in Appendix B. (Recall that a + b means "a OR b" and that a · b means "a AND b.")

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$

If a · b · CarryIn is true, then all of the other three terms must also be true, so we can leave out this last term corresponding to the fourth line of the table. We can thus simplify the equation to

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

Figure 4.13 shows that the hardware within the adder black box for CarryOut consists of three AND gates and one OR gate. The three AND gates correspond exactly to the three parenthesized terms of the formula above for CarryOut, and the OR gate sums the three terms.

Inputs		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

FIGURE 4.12 Values of the inputs when CarryOut is a 1.

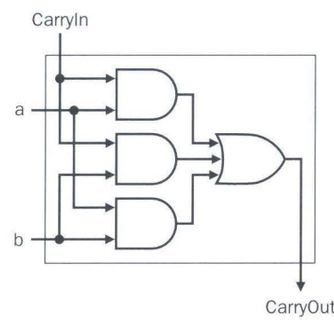


FIGURE 4.13 Adder hardware for the carry out signal. The rest of the adder hardware is the logic for the Sum output given in the equation on page 234.

The Sum bit is set when exactly one input is 1 or when all three inputs are 1. The Sum results in a complex Boolean equation (recall that $\bar{a}$ means NOT a):

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

The drawing of the logic for the Sum bit in the adder black box is left as an exercise (see Exercise 4.43).

Figure 4.14 shows a 1-bit ALU derived by combining the adder with the earlier components. Sometimes designers also want the ALU to perform a few more simple operations, such as generating 0. The easiest way to add an operation is to expand the multiplexor controlled by the Operation line and, for this example, to connect 0 directly to the new input of that expanded multiplexor.

A 32-Bit ALU

Now that we have completed the 1-bit ALU, the full 32-bit ALU is created by connecting adjacent "black boxes." Using *xi* to mean the *i*th bit of *x*, Figure 4.15 shows a 32-bit ALU. Just as a single stone can cause ripples to radiate to the shores of a quiet lake, a single carry out of the least significant bit (Result0) can ripple all the way through the adder, causing a carry out of the most significant bit (Result31). Hence, the adder created by directly linking the carries of 1-bit adders is called a *ripple carry* adder. We'll see a faster way to connect the 1-bit adders starting on page 241.

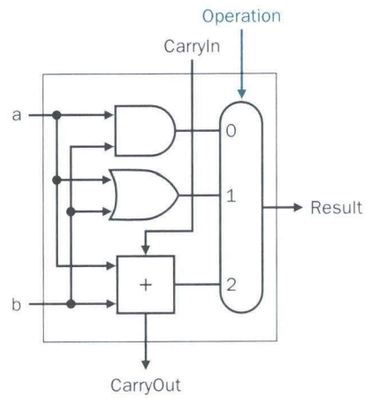


FIGURE 4.14 A 1-bit ALU that performs AND, OR, and addition (see Figure 4.13).

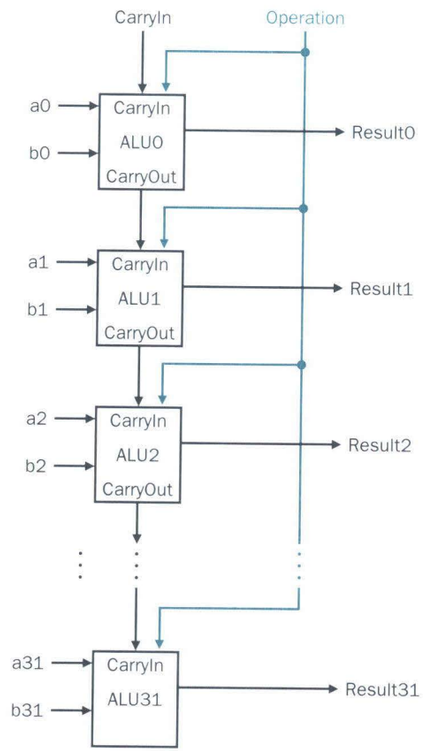


FIGURE 4.15 A 32-bit ALU constructed from 32 1-bit ALUs. CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.

Subtraction is the same as adding the negative version of an operand, and this is how adders perform subtraction. Recall that the shortcut for negating a two's complement number is to invert each bit (sometimes called the *one's complement* as explained in the elaboration on page 219) and then add 1. To invert each bit, we simply add a 2:1 multiplexor that chooses between b and $\bar{b}$, as Figure 4.16 shows.

Suppose we connect 32 of these 1-bit ALUs, as we did in Figure 4.15. The added multiplexor gives the option of b or its inverted value, depending on Binvert, but this is only one step in negating a two's complement number. Notice that the least significant bit still has a CarryIn signal, even though it's unnecessary for addition. What happens if we set this CarryIn to 1 instead of 0?