CS 61C Lecture Notes Week 5

Topic: Procedures and stack, arithmetic

Reading: Patterson & Hennessy, Sections 3.6, 4.1–4, 4.6–7

Procedures

What pieces of information must be communicated between caller and callee when one procedure invokes another?

- The starting address of the callee.
- The address in the caller to which to return.
- The arguments.
- The return value.

In addition, for each invocation we must allocate memory for

- the called procedure's local variables.
- any registers that must be saved to avoid conflicting use.

The extent to which these needs are met by special procedure-call hardware varies tremendously among architectures. In principle there is no need for any instructions, beyond the ones we've already seen, just for calling procedures. The Vax, the textbook example of a CISC architecture, had a very complicated procedure calling instruction that does almost all of the things listed above, all in one instruction. The PDP-10 had four different procedure calling instructions, with varying degrees of hardware support. (For example, one of them doesn't use any registers, but doesn't allow for recursion.)

The MIPS provides *minimal* hardware support with the `jal` (Jump And Link) instruction. It's like an ordinary jump instruction (so it's where you give the starting address of the callee), but it also saves the return address in the caller, which is the only thing on the list that would be difficult with other instructions. Recall that the MIPS processor has a special register, the PC or Program Counter, that contains the address of the next instruction to be fetched; this is exactly the address to which the procedure should return when it's finished. But the PC is not one of the 32 general registers, and so its value can't be saved with instructions like `sw`. What `jal` does is to copy the PC into `$31` (`$ra`), and then set the new value of the PC from the address operand in the instruction. This is one of the two instances in the MIPS hardware of a special property of one of the general registers. (The other is that `$0` is always zero.)

How does the procedure actually use the address in `$31` to return to its caller? It uses the instruction

```
jr $31
```

which means Jump to (the address in) Register 31.

Arguments and return values

The procedure calling convention used in the MIPS machines is to use specific registers for arguments and return values. In particular, registers `$4` through `$7` (`$a0`–`$a3`) are reserved for arguments, and register `$2` (`$v0`) is reserved for the return value. (Just in case a procedure returns a 64-bit value, such as a double-precision floating point number, `$3` (`$v1`) is also reserved for possible use in a return value. We won't need that rule in this course, though.) So the procedure call

```
x = foo(87, 24, 5)
```

will be translated as

```
li   $4, 87
li   $5, 24
```

```
        li   $6, 5
        jal  foo
        sw   $2, x
```

Of course this convention doesn't work if a procedure takes more than four arguments, or if some argument or return value is too big to fit in a register. An alternate convention, which we'll see later, is used in those cases.

Note that these specific register uses are *not* built into the hardware, the way the special roles of `$0` and `$31` are. These are just arbitrary software conventions.

Stack frames

We still need to find room in memory for a procedure's local variables and other temporary storage. For this purpose we use the *stack*. Whenever a procedure is invoked, the called procedure knows how much temporary memory this invocation needs, and it allocates that amount of stack space. How is this allocation done? By convention, register `$29` (also called `$sp`) is used as the *stack pointer*. It contains the lowest address currently in use on the stack. (Note: P&H contradicts itself on this point; sometimes it says that `$sp` contains four less than that: the highest address *not* currently in use. But we'll use the former convention.)

To allocate, say, 24 bytes of stack space, a procedure merely subtracts 24 from the stack pointer:

```
foo:    addi  $29, $29, -24
```

(Remember that there is no `subi` instruction in the MIPS.) To release that space at the end of the procedure, it adds 24 to the pointer. These 24 bytes of stack space are called the procedure's *stack frame*.

Stack space is always allocated in multiples of a word (i.e., in amounts divisible by four). That's so that another procedure called by this one can assume that the address in `$sp` can be used as a word address.

Variables declared local to a procedure are part of the procedure's stack frame. For example, in compiling this procedure:

```
int foo(int a, int b) {
    int x;
    int array[10];
    int y;

    ...
}
```

the C compiler will determine that the procedure needs 12 words (48 bytes) of stack frame. Then it might use `0($29)` for `x`, `4($29)` for `array`, and `44($29)` for `y`. In other words, the local variables are addressed using a base/offset notation in which the stack pointer is the base register.

You've heard the word "frame" before in connection with procedure calling, namely, in 61A. This is not a coincidence! The environment frames in 61A contain the values of local variables, and so do the stack frames in 61C.

There is one difference, though. In C, suppose that we have a chain of procedure calls, with procedure A calling procedure B, which calls procedure C, which calls D, which calls E. At this point there are five frames on the stack, with the one for the invocation of A at the top, and the one for E at the bottom. It should be clear that procedure E will be the first to return; when it releases its stack frame, the ones for A through D will still be on the stack. Then D can return, and so on. At any moment, no matter how many calls and returns have happened, the frames for the currently active procedures will be together at the top end of the stack.

That's not quite true in Scheme. The reason is that in Scheme the frame for a procedure invocation may still be needed after that procedure returns! This will be the case if the procedure returns a procedure, like this:

```
(define (make-adder num)
  (lambda (x) (+ x num)))
```

After we say

```
(define plus3 (make-adder 3))
```

the call to `make-adder` has returned, but we still need its frame so that `plus3` will remember that it uses the value 3 for `num`. This is how we implemented object-oriented programming, with local state variables, in Scheme.

C can't do this because procedures are not first-class data in C. That is, a C procedure can't create and return another procedure. So there's no way we can still need a procedure invocation's frame after the procedure has returned. This is why C people felt they had to invent a special extension language (C++) in order to do object-oriented programming, whereas in Scheme we can do OOP merely by nesting `lambda`s.

We say that a language, such as C, in which a frame can be discarded when the procedure returns "obeys stack discipline."

Note that I've been talking about "a procedure *invocation*'s frame," not a procedure's frame. Each invocation gets its own frame; otherwise we couldn't have recursive procedures in which each invocation has its own values for local variables.

Saving and using registers

Since the `jal` instruction saves the PC into `$31`, what happens when one procedure invokes another? The return address for the second call replaces the return address for the first call. This means that the first procedure no longer knows where to return.

The solution to this problem is that a procedure that intends to call another procedure must save `$31` on the stack when it begins, and restore the saved value when it wants to return. Here's a very simple procedure that invokes another procedure:

```
int foo(void) {
    return baz()+3;
}
```

And here's its translation into MAL:

```
foo:    addi $29, $29, -4       # prologue
        sw   $31, 0($29)

        jal  baz                # body
        addi $2, $2, 3

        lw   $31, 0($29)        # epilogue
        addi $29, $29, 4
        jr   $31
```

Every procedure begins with a *prologue* in which it allocates a stack frame and saves any necessary registers. (We'll see in a moment that there might be others besides `$31`.) Then comes the *body* that actually does the computation described by the higher-level language programmer. Finally, the procedure ends with an *epilogue* in which it restores registers, deallocates the stack frame, and returns to the caller.

Besides `$31`, procedures use registers for two reasons. We've often seen that registers are needed for short-term, temporary use in the process of a computation. For example, a C statement like

```
x = (a+b) - (c+d);
```

will be compiled into MAL instructions using three temporary registers:

```
lw    $8, a
lw    $9, b
add   $8, $8, $9
lw    $9, c
lw    $10, d
add   $9, $9, $10
sub   $8, $8, $9
sw    $8, x
```

The other reason is that it's often faster to keep some local variables in registers, rather than in the stack frame. If a variable is used several times, keeping it in a register avoids many load and store instructions.

Each procedure is compiled independently. If one procedure calls another, there is the possibility of a conflict in which both use the same register. If the caller expects that register to have the same value after the call as it did before the call, it will compute an incorrect result.

Clearly the solution is that one of the two procedures must save and restore any registers for which this collision is possible. Should it be the caller or the callee that takes this responsibility? Both answers are possible. "Caller saves" means that a procedure may make no assumptions about the values in registers after a call; the caller must save any important register values before the call and restore them after it. "Callee saves" means that a procedure may assume that the values in registers will be preserved when it calls another procedure; in this case it is the called procedure that must save and restore any registers it uses.

Why does it matter which we choose? The goal is to avoid unnecessary load and store instructions. For example, in a system using caller saves, the caller might needlessly save several registers that the callee happens not to use. But in callee saves, the callee might save and restore registers that the caller doesn't need.

The MIPS solution is a compromise. Registers `$8` through `$15` (`$t0`–`$t7`) are *temporary* registers, used to hold intermediate values during the computation of a C expression. For these registers, the convention is that a called procedure may change their values without saving and restoring the old values; the caller may not assume anything about these registers after a procedure call. But registers `$16` through `$23` (`$s0`–`$s7`) are *saved* registers, used for local variables whose value will be needed throughout the body of the procedure. Each procedure must save and restore the values of whichever of these registers it uses.

Which local variables should be kept in the stack frame, and which in registers? Obviously a variable that's bigger than a word, such as an array, can't be in a register. Beyond that, it's up to the compiler to decide. C includes a `register` keyword with which the programmer can give the compiler advice, but the compiler is not required to follow the advice. Early compilers did generally follow the user's suggestions, but today's optimizing compilers can do a better job than people can, and so most compilers pay no attention to the `register` keyword.

Frame pointers

The convention we're using is that a procedure allocates all the stack space it will need, right at the beginning of its prologue. This simple rule works fine for C programs. But it's also possible to use a more complicated approach in which a procedure may allocate additional stack space in the middle, as the space is needed.

Allocating stack space means changing the value in `$29`. If a procedure does that, the offsets needed to find its local variables will change. This is undesirable. Therefore, some languages or compilers use *two* pointers into the stack. The stack pointer points to the first free position; the *frame pointer* (`$30` or `$fp`) points to the frame that the procedure allocated in its prologue. References to local variables then use the frame pointer as the base register, rather than the stack register.

But don't use this complexity in this course. For our purposes, procedures will always allocate all their stack space in the prologue. I'm only telling you about the frame pointer so you don't get confused if you read other MIPS literature.

Arguments on the stack

If an argument is too big to fit in a register, it must be passed on the stack. This is a little tricky because, of course, it's the *caller* that must put the value on the stack, so the space must be part of the caller's frame. But the callee has to know where to find it.

The convention is this: Stack arguments must be at the bottom of the caller's frame. The first argument is at the very bottom, then the second, and so on. Here's a simple example:

```
int cequal(struct complex a, struct complex b) {
    int requal, iequal;

    requal = (a.real == b.real);
    iequal = (a.imag == b.imag);
    return requal && iequal;
}
```

I've deliberately used the unnecessary local variables `requal` and `iequal` so that this procedure will have its own stack frame. We'll put those variables in `$16` and `$17`, but the old values in those registers must then be saved on the stack.

```
cequal: addi  $29, $29, -8
        sw    $16, 0($29)
        sw    $17, 4($29)

        lw    $8, 8($29)          # a.real
        lw    $9, 16($29)         # b.real
        beq   $8, $9, ryes
        add   $16, $0, $0         # not equal
        b     rdone
ryes:   addi  $16, $0, 1          # equal
rdone:  lw    $8, 12($29)         # a.imag
        lw    $9, 20($29)         # b.imag
        beq   $8, $9, iyes
        add   $17, $0, $0
        b     idone
iyes:   addi  $17, $0, 1
idone:  beqz  $16, no
        beqz  $17, no
        addi  $2, $0, 1
        b     epilog
no:     add   $2, $0, $0
epilog: lw    $16, 0($29)
        lw    $17, 4($29)
        addi  $29, $29, 8
        jr    $31
```

The crucial point here is that `cequal`'s stack frame is only eight bytes long, but the body makes reference to words as far up as 20 bytes. Those are references to the caller's stack frame. The complex argument `a` is in the two words starting at `8($29)`, just above `cequal`'s frame; the argument `b` is in the two words above that, starting at `16($29)`.

Since the space for the arguments was allocated by the caller, it will be deallocated by the caller; `cequal` deallocates only the eight bytes that it allocated in its prologue.

What if the return value doesn't fit in a register? In that case, the caller allocates stack space for that, too; this space comes at the bottom, as a sort of argument zero, before the first argument.

What if some arguments fit in registers and others don't? Never mind; in this course we'll almost always write programs in which all arguments and return values fit in registers. The only exception will be the `sprintf` project, in which all arguments will go on the stack, but the return value will still be in `$2`. Of course a real compiler will have to have conventions for all possible cases.

Arrays as arguments and return values

We've talked about this already from a C standpoint, but it's worth revisiting the question from a MAL point of view.

As an example, I'd like to add the elements of two arrays pairwise to form a third array. Let's suppose the arrays are 100 integers long.

First version, caller passes output array by reference:

```
void sumarrays(int a[], int b[], int c[]) {
    int i;

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

Since the procedure has three arguments, and the array arguments are passed by reference (that is, what's really passed is a pointer to the array), all the arguments fit in registers:

```
sumarrays:
        add   $8, $0, $0            # i=0
loop:   bge   $8, 100, fini         # done if i >= 100
        lw    $9, 0($4)             # a[i]
        lw    $10, 0($5)            # b[i]
        add   $9, $9, $10
        sw    $9, 0($6)             # c[i]
        addi  $8, $8, 1             # i++
        addi  $4, $4, 4             # increment pointers
        addi  $5, $5, 4             # (optimizing compiler!)
        addi  $6, $6, 4
        b     loop
fini:   jr    $31
```

Don't return local arrays

This version works fine, as long as the caller has allocated space for all three arrays and passes pointers to that space as arguments. The only thing wrong with it is that since it isn't functional, it can't be composed. In other words, if we want to add three arrays we have to say

```
sumarrays(a, b, x);
sumarrays(x, c, y);
```

using an intermediate array `x`, whereas we'd like to say

```
y = sumarrays(sumarrays(a, b), c);
```

just as we could if we were adding individual numbers.

So here's my next attempt:

```
int *sumarrays(int a[], int b[]) {           /* wrong! */
    int i;
    int c[100];

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

The C compiler won't complain about this procedure, and will cheerfully compile it into this (using `$8` for `i`, but putting the array `c` on the stack):

```
sumarrays:
        addi  $29, $29, -400
        add   $8, $0, $0            # i=0
        add   $11, $29, $0          # pointer to array c
loop:   bge   $8, 100, fini         # done if i >= 100
        lw    $9, 0($4)             # a[i]
        lw    $10, 0($5)            # b[i]
        add   $9, $9, $10
        sw    $9, 0($11)            # c[i]
        addi  $8, $8, 1             # i++
        addi  $4, $4, 4             # increment pointers
        addi  $5, $5, 4             # (optimizing compiler!)
        addi  $11, $11, 4
        b     loop
fini:   add   $2, $29, $0           # return value is pointer to c
        add   $29, $29, 400         # deallocate frame
        jr    $31
```

What's wrong with this procedure? It returns a pointer to a block of memory, the array `c`, that's part of its own stack frame. But it deallocates the stack frame before it returns! That won't matter right away, because the deallocated memory still has the correct values in it, but as soon as we call another procedure it'll be overwritten. For example, if we try to print the values in the array `c`, some of the ones near the end won't be correct.

The solution is to heap-allocate the array instead of stack-allocating it:

```
int *sumarrays(int a[], int b[]) {
    int i;
    int *c;

    c = (int *)malloc(100 * sizeof(int));
    for (i=0; i<100; i++)
        c[i] = a[i] + b[i];
    return c;
}
```

Since `sumarrays` now invokes a procedure, `malloc`, we have to be more careful about using registers. We must save `$31` and use saved registers, rather than temporary registers, to hold the variables.

```
sumarrays:
        addi  $29, $29, -20
        sw    $31, 0($29)
        sw    $16, 4($29)           # i
        sw    $17, 8($29)           # c
        sw    $4, 12($29)           # save arguments too
```

```
        sw    $5, 16($29)

        addi  $4, $0, 400
        jal   malloc                # allocate array c
        add   $17, $2, $0           # address returned in $2
        add   $11, $17, $0          # incrementable pointer to array c
        lw    $4, 12($29)           # restore args
        lw    $5, 16($29)

        add   $16, $0, $0           # i=0
loop:   bge   $16, 100, fini        # done if i >= 100
        lw    $9, 0($4)             # a[i]
        lw    $10, 0($5)            # b[i]
        add   $9, $9, $10
        sw    $9, 0($11)            # c[i]
        addi  $16, $16, 1           # i++
        addi  $4, $4, 4             # increment pointers
        addi  $5, $5, 4             # (optimizing compiler!)
        addi  $11, $11, 4
        b     loop

fini:   add   $2, $17, $0           # return value is pointer to c
        lw    $31, 0($29)
        lw    $16, 4($29)
        lw    $17, 8($29)
        add   $29, $29, 20          # deallocate frame
        jr    $31
```

Understanding how stack frames are allocated and deallocated makes comprehensible the otherwise strange C rule that you can't return a pointer to a local array. Heap-allocated arrays are first-class but stack-allocated arrays aren't.

Scalar vs. aggregate arguments

When an array is used as an argument in C, it's the *address* of the array that goes in the argument register. But when a scalar (a single number) is used as an argument, it's the *value* that goes in the register. For example, suppose we want to add the same value to every element of an array. For simplicity I'll go back to the first idea in which the output array is passed by the caller.

```
void shiftarray(int a[], int x, int c[]) {
    int i;

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

In the MAL translation, watch how `a[i]` must be fetched from memory, as before, but `x` is already in a register:

```
sumarrays:
        add   $8, $0, $0            # i=0
loop:   bge   $8, 100, fini         # done if i >= 100
        lw    $9, 0($4)             # a[i]
        add   $9, $9, $5            # a[i] + x
        sw    $9, 0($6)             # c[i]
        addi  $8, $8, 1             # i++
```

```
        addi  $4, $4, 4             # increment pointers
        addi  $6, $6, 4             # (not including $5!)
        b     loop
fini:   jr    $31
```

Static variables

Another solution to the problem of disappearing arrays is to declare them `static`. This is the same idea as the local state variables in 61A; the variable belongs to one procedure, but is permanently allocated and retains its value between invocations, instead of being allocated in the stack on each invocation. Static variables are allocated as if they were global variables, in the data area below the heap. So we can say

```
int *sumarrays(int a[], int b[]) {
    int i;
    static int c[100];

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

and this version returns a pointer to an array that won't disappear, but, as discussed earlier, will be over-written on a second call.

```
sumarrays:
        add   $8, $0, $0            # i=0
        la    $11, c                # c has a fixed address
loop:   bge   $8, 100, fini         # done if i >= 100
        lw    $9, 0($4)             # a[i]
        lw    $10, 0($5)            # b[i]
        add   $9, $9, $10
        sw    $9, 0($11)            # c[i]
        addi  $8, $8, 1             # i++
        addi  $4, $4, 4             # increment pointers
        addi  $5, $5, 4             # (optimizing compiler!)
        addi  $11, $11, 4
        b     loop
fini:   la    $2, c                 # return value is pointer to c
        jr    $31
```

Logical operations

In C (and friends) there are the familiar *logical* or *Boolean* operators `&&` (and), `||` (or), and `!` (not), used to compute true/false conditions for use with `if`, `while`, etc.

There is also a related but different set of *bitwise* operators: `&` (bitwise and), `|` (bitwise or), and `~` (bitwise not).

The value produced by a C logical operator is always either the integer 0, meaning false, or the integer 1, meaning true. For the bitwise operators, the result can be any integer; what they do is match each bit of one operand with the corresponding bit of the other, and make the corresponding bit of the result be the appropriate logical function of those bits. For example:

```
(1100 binary) && (1010 binary) ===> 1

(1100 binary) & (1010 binary) ===> (1000 binary)
```

because the second value is computed this way:

```
        1 1 0 0
and     1 0 1 0
      ---------
        1 0 0 0
```

Flags

One use of bitwise operations is to allow operations on sets with a small number of elements. For example, here is a sketch of a calendar program:

```
#define MON 0x01
#define TUE 0x02
#define WED 0x04
#define THU 0x08
#define FRI 0x10
#define SAT 0x20
#define SUN 0x40

int classes = MON|WED|FRI;
int officehours = MON|THU;

int busy;

busy = classes & officehours;
```

The value of the variable `busy` will be the set of days on which I have both classes and office hours (namely, `MON`).

Things to note here: First, the values chosen for the days of the week can't be any old numbers; each of them is a power of two, so that there is a single one bit and the rest of the bits are zero.

Second, in forming the set variables `classes` and `officehours`, why do I use the *or* operation instead of *and*? You might expect the latter because in English we say "I teach classes Monday *and* Wednesday *and* Friday." But do the computation; the bitwise *and* of `MON`, `WED`, and `FRI` is zero, whereas the bitwise *or* of those values is `0x15` or binary 0010101. Each one bit represents a member of the set.

In computing the value of `busy` I use the bitwise *and* to find the intersection of two sets. This computation does follow ordinary English usage; it's the set of days on which I have both classes *and* office hours.

The MIPS hardware provides `and`, `or`, and `not` instructions for these bitwise operators. (There is also an `xor` instruction for the exclusive-or function as described in P&H.) There are also immediate versions `andi` and `ori`. (It wouldn't make sense to have a `noti` because `not` only uses one argument value.) As in the immediate instructions for arithmetic, the instruction includes a 16-bit immediate value that must be extended to 32 bits. But unlike the case of arithmetic operations, in these instructions the arguments are not considered to be integers; they are sets of 32 independent bits, as described above. Therefore, it makes no sense to sign extend the immediate operand; the logical immediate instructions always use zero as the left half of the immediate operand.

Translating logical operators to MAL

Here's a (rather useless) fragment of C code:

```
int a,b,x,y;

a = x && y;
b = x & y;
```

The Boolean operator is translated by case analysis; the bitwise operator takes a single machine instruction.

```
        lw   $8, x
        lw   $9, y
        beqz $8, zero         # beginning of && computation
        beqz $9, zero
        addi $10, $0, 1       # get here if both are nonzero
        b    fini
zero:   add  $10, $0, $0      # get here if either is zero
fini:   sw   $10, a
        and  $11, $8, $9      # entire & computation
        sw   $11, b
```

Shift operators

In addition to the bitwise logical operations, there's another thing we might want to do with the bits in a word: move them over to the left or right.

For example, suppose we have a machine language instruction in R-format and we want to know the value of its source register field:

```
+------+-----+-----+-----+-----+------+
|opcode|  rs |  rt |  rd |shamt| funct|
|   6  |  5  |  5  |  5  |  5  |   6  |
+------+-----+-----+-----+-----+------+
        ^^^^^
        We want these five bits.
```

We can use a bitwise `and` instruction to turn off (set to zero) all the undesired bits:

```
        lw    $8, instr          # the instruction word
        li    $9, 0x03e00000
        and   $10, $8, $9
```

Where did that `0x03e00000` come from? Well, make a 32-bit binary number by setting the five bits in the `rs` field to one, and all the other bits to zero:

`000000 11111 00000 00000 00000 000000`

Then take those bits four at a time, to convert to hexadecimal, and you'll have the magic number.

The problem is that the result won't quite be what we want, because we want the register number as an integer between 0 and 31, and for that we must have those five bits in the rightmost five bits of the result. So after doing the `and` we can add one more instruction:

```
        srl   $10, $10, 21
```

This says to take the value in register 10, shift it 21 bits to the right, and put the result back in register 10. When we shift, the 21 rightmost bits of the original value are lost; the 21 leftmost bits of the result will be zero.

The name `srl` stands for Shift Right Logical; there is also a Shift Left Logical (`sll`) and a Shift Right Arithmetic (`sra`). The meaning of `sll` should be obvious, but `sra` may require some explanation.

Shifting a binary number to the left involves writing some extra zeros on its right end. Just as writing a zero at the end of an ordinary decimal integer multiplies the value by ten, writing a zero at the end of a binary integer multiplies it by two. Shifting left by N bits multiplies the value by 2^N. For example, we've occasionally needed to multiply an array index by 4 because the elements of the array are integers and we want the byte address of an element. Instead of using a `mul` instruction, the compiler would really use `sll` with a shift count of 2.

If shifting left multiplies by a power of 2, what does shifting right do? We'd expect it to *divide* by a power of 2. This works fine if the original value is positive, although of course if the original value isn't divisible by 4 the result is truncated to the next lower integer, as it would be in an integer divide instruction. But things are a little trickier if the number is negative, for two reasons. One reason is roundoff; if we divide a negative number by four, should the result be truncated to the next lower value, or to the negative of the next lower magnitude? For example, should -3.5 be represented as -4 (next lower integer) or as -3 (next lower absolute value)? Right now I'm more interested in the other complication, which is that the result won't be a negative integer unless the leftmost bits of the result are set to *one*, not to zero. For example, here is the number -16 as a 32-bit binary integer:

```
11111111111111111111111111110000
```

If we divide by four, by shifting two places to the right, we get two possible answers depending on how we fill those two leftmost bits:

```
00111111111111111111111111111100       srl
11111111111111111111111111111100       sra
```

The second of these is the representation of -4, which is the answer we want if we're using the shift to do arithmetic. On the other hand, if we're using the shift to isolate some of the bits, as in my example about the source register field of an instruction, we want to shift in zeros, not ones. That's why we have two kinds of right shift. (Why don't we need two kinds of left shift?)

Immediate vs. variable operands

For arithmetic, the usual thing is to find both argument values in registers (e.g., `add`), and we have a special name (`addi`) for the case in which one of the arguments has a fixed value encoded in the instruction itself.

For shifting, the situation is reversed. Most of the time, the programmer always wants to shift some value by a constant number of bits, as in the case of conversion between word and byte addresses, or the 21 that was built into my `rs` finder above. Only occasionally is the shift amount variable. That's why the MIPS uses the short names `sll` and so on for the fixed-amount shift instructions, and provides longer names `sllv`, `srlv`, and `srav` for the variable-amount shifts. These instructions have three register operands, like ordinary arithmetic instructions.

If `sll`, `srl`, and `sra` have a fixed operand, do they use I-format? No, they are still R-format, because the only reasonable shift amounts are between 1 and 31. (If we shift by more than 31 bits, all of the data in the original word are gone, and we have a word full of zeros or full of ones.) This is what the `shamt` field in the R-format instructions is for. It's used only in the `sll`, `srl`, and `sra` instructions.

How to multiply integers

We aren't going to worry a lot about the detailed hardware arithmetic algorithms, but just to show you they're not as scary as you might think reading P&H, here's how we multiply integers.

First, recall how you do it in plain old decimal integers:

```
      472
    x 153
---------
     1416
    2360
    472
---------
    72216
```

The key point here is that there's a partial product for each digit of the multiplier, and the partial products are shifted over according to which digit it is, and they're all added.

Exactly the same algorithm works in binary, except that each bit of the multiplier is either zero or one; if the bit is zero, the partial product is zero, and if the bit is one, the partial product is equal to the multiplicand. So there's no need to do any actual multiplying! It's all shifting and adding:

```
     110101
   x  10011
-----------
     110101
    110101
 110101
-----------
 1111101111
```

(I haven't bothered writing down the partial products that are zero.)

So the algorithm is this:

1. Product starts out zero.

2. If multiplier is zero, we're done.

3. Look at last bit of multiplier. If it's one, add multiplicand to product.

4. Shift multiplier one bit rightward.

5. Loop to step 2.

The product register has to be double-width to hold the largest possible product; that's the only real complication. All of the refinements P&H discuss are just storage optimizations to save a few bits of hardware.