CS 61C Lecture Notes Week 4

Topic: Machine language, pointers

Reading: Patterson & Hennessy, Chapter 3 (except 3.6) and appendix A.9, A.6

Machine language is the representation of a program that the computer hardware directly understands. A machine language program is a sequence of *instructions*. *Assembly language* is a human-readable representation of a program in which each instruction directly corresponds (except for exceptions, noted below) to one machine language instruction.

In the MIPS architecture, arithmetic instructions have a *three-operand* format. That is, in each instruction you must specify two arguments to the arithmetic function and a place to put the result. (In some *two-operand* architectures, the result always goes into wherever the first argument comes from.)

In some really old memory-based machines, the three operands of an arithmetic instruction are in the computer's main memory. But in the MIPS machines, as in all modern computers, the arithmetic operands must be in a set of *registers* (32 of them in the MIPS) that are part of the central processor—in effect, another very small memory. The registers are identified by number, from 0 to 31. (We'll see later that some of those registers have special purposes, which is why most of my examples use register numbers starting at 8. For example, register 0 always contains the value zero; you can't put a different value into it.) We refer to registers by symbols starting with a dollar sign, followed either by the register number, as in `$8`, or by a name indicating the conventional use for the register, as in `$t0` for "temporary register 0," which is the name of `$8` because it's used for holding temporary values in the middle of a computation. Register `$0`'s name is `$zero`.

What's the point of registers? The reason is efficiency. First, for any given technology, a small memory can be made faster than a large one. Second, it's cost-effective to use more expensive technology for a small memory than we could afford for a large one. Third, since the registers are inside the processor, they are not limited by the memory bus bottleneck; the processor's wiring allows access to more than one register at a time. (This third reason is probably the most important.) In addition to the bottleneck, the memory bus also slows things down because of its length; today's technology is fast enough so that the propagation speed of electrons through wire significantly affects the overall speed of the machine.

Instruction formats

Each machine language instruction is represented as a pattern of 32 bits, just as integers are, but of course with a different encoding. Essentially an instruction is a bunch of small unsigned integers packed into one 32-bit value. For example, the assembly language instruction

```
add $10, $8, $9
```

(which says to add the values in register number 8 and register number 9, putting the result into register number 10) is represented in machine language using this bit pattern:

```
00000001000010010101000000100000
```

That unreadable mess becomes more comprehensible if we break it into pieces this way:

```
000000 01000 01001 01010 00000 100000
```

The `000000` at the left is the *opcode*; in this case, `000000` indicates an arithmetic instruction, further specified by the six-bit *function* field at the right. In this case, `100000` means the `add` instruction.

The three five-bit fields `01000`, `01001`, and `01010` represent the three register numbers in the instruction. The first two are the operand registers, and the third indicates the destination register for the result. The remaining five-bit field `00000` isn't used in the `add` instruction, but is used in some other instructions that we'll see later.

The `add` instruction is an example of what the MIPS designers call an *R-format* instruction. It's divided into six fields, as shown above; here are their names and widths (in bits):

```
+------+-----+-----+-----+-----+------+
|opcode|  rs |  rt |  rd |shamt| funct|
|   6  |  5  |  5  |  5  |  5  |   6  |
+------+-----+-----+-----+-----+------+
```

`Rs`, `rt`, and `rd` represent the *source* register, the *target* register, and the *destination* register. The names "source" and "destination" are probably obvious; the name "target" is used because, depending on the instruction, this can be either a source register (as in the `add` instruction) or a destination register.

Note that the order of the register operands in the assembly language instruction is different from the order in the machine language instruction. The assembly language has the destination register on the left, then the source, then the target. (The order of source and target doesn't matter for `add`, which is commutative, but it does matter for other arithmetic instructions such as `sub`.) The theory behind the assembly language order is that it matches the order you'd use in a higher-level language assignment statement:

```
reg10 = reg8 + reg9;
```

The order of operands in the machine language instruction (which isn't meant to be read or written by human beings at all, so it doesn't have to be mnemonic) is chosen so that the `rs` and `rt` fields are in the same place as they are in the MIPS *I-format* instructions, which look like this:

```
+------+-----+-----+----------------+
|opcode|  rs |  rt |   immediate    |
|   6  |  5  |  5  |      16        |
+------+-----+-----+----------------+
```

The "immediate" field allows an operand to be included within the instruction itself. For example, if we always want to add the value 87 to something, we can say

```
addi  $10, $8, 87
```

which will be translated into the machine language instruction

```
001000 01000 01010 0000000000010111
```

The leftmost `001000` is the operation code for `addi`, which means "add immediate." The `01000` is the source register, 8. The `01010` is the target register, 10, here used as a destination. And the 16-bit pattern at the end represents the value 87. This 16-bit value is used as a signed, twos-complement integer.

[Because a string of 32 ones and zeros is very hard for a human being to read, when we want to represent binary numbers we usually use *hexadecimal* as an abbreviation. The bit pattern for our example `add` instruction has a hex value of `01095020`. (Make sure you see how each hex digit corresponds to a group of four bits.) We will generally follow the C convention of writing `0x` in front of hexadecimal values, so for example `0xc` represents 12 (decimal).]

The handling of negative immediate operands is slightly tricky. In order to use a 16-bit immediate operand along with a 32-bit register value, we must extend the 16 bits to 32. For positive values, this is easy; we use the immediate operand for the rightmost 16 bits, and set the leftmost 16 bits to zero. But for a negative value, because of the way twos complement works, we must set the leftmost 16 bits to *one*! The general rule is that the leftmost 16 bits of the 32-bit value should be equal to the leftmost bit of the given 16-bit operand. This process is called *sign extension*.

(The MIPS has an instruction called `addiu`, for Add Immediate Unsigned. You might think that that instruction would zero-extend the immediate operand instead of sign-extending it, since it is to be considered as an unsigned value. But no, that's not what it means. This badly named instruction, just like the equally badly named `addu` instruction, does not affect the handling of its operands. The only difference is that the "Unsigned" instructions do not signal the processor in the event of an arithmetic overflow. That's why C compilers always use `addu` and friends; since C has no mechanism for catching an overflow indication, there is no benefit to letting the hardware generate one.)

Load/store instructions and memory addresses

How do the operands get into the registers in the first place? Besides the arithmetic instructions, the MIPS architecture also includes *load* instructions (to get information from memory into a register) and *store* instructions (from register to memory). For example, the instruction

```
lw $8, 100
```

loads the word of memory at address 100 into processor register 8.

(A *word* of memory is 32 bits. We'll also sometimes deal with a *byte* of memory, which is 8 bits.)

Memory addresses

All of a program's instructions and data live in memory. How does the computer find the particular piece of memory it needs next? The answer is that each piece of memory has an *address*, just like the address of a house on a street. The processor sends an address to the memory, and the memory sends the data at that address back to the processor.

What's a "piece" of memory? In the MIPS architecture, a piece can be a *byte* (8 bits), a *halfword* (16 bits), or a *word* (32 bits). Bytes are generally used to represent text characters; words are used for pretty much everything else. We won't use halfwords at all in this course. The idea of a halfword is left over from the days when memory was expensive and it was important to use as little as possible.

The size of a machine's word is determined by the width of its bus—that is, by the number of wires in the bus devoted to data. (Exception: The Intel 8088 processor, used in the original IBM PCs, had a 16-bit word but an 8-bit bus. Each transfer over the bus took two cycles. This was a kludge to try to fit new processor technology into old bus technology.)

A memory address is an unsigned integer, so the first byte of memory has address zero. The MIPS is *byte-addressible*, which means that each byte has its own address. So the maximum size of memory is 2^{32} bytes, which is about four billion. Few actual computers have that much main memory today, although disks have surpassed that size.

What about the address of a word? A word is four bytes. The rule in the MIPS, as in most computers, is that they can't be any old four bytes; they must be four consecutive bytes, the first of which has a multiple of four as its address. So, for example, bytes 12, 13, 14, and 15 form a word, but bytes 14, 15, 16, and 17 do not. The address of a word is the address of its first byte.

(Note: At the beginning of chapter 3, Patterson & Hennessy pretend that words have consecutive addresses: 0, 1, 2, etc. Later in the chapter they let on that word addresses are really multiples of four: 0, 4, 8, etc. Don't let this confuse you.)

Memory and registers

A memory address is 32 bits wide. An instruction is also 32 bits wide. This means that a memory address can't fit into an instruction. So how do the load and store instructions, which have a memory address as an operand, work?

The load and store instructions are in I-format. They use a combination of the 16-bit immediate field and a register, the one indicated in the RS field, to compute the desired memory address. The immediate value (a signed 16-bit integer) is added to the value in the register to form the desired address. In this context, the immediate value is called an *offset*, and the register is called a *base register*.

Consider the following fragment of a C program:

```
int i;
int a[10];
```

```
for (i=0; i<10; i++)
    a[i] = 0;
```

If we happen to know that the array `a` will be stored at memory addresses less than 2^{15}, then we could translate the program to MIPS assembly language this way:

```
        add  $8, $0, $0
loop:   bge  $8, 40, fini
        sw   $0, a($8)
        addi $8, $8, 4
        b    loop
fini:
```

To understand this program, remember that register 0 always contains zero. So the first instruction sets register 8 to zero. The second instruction says that we're finished with the loop if register 8 is greater than or equal to 40. The third instruction stores zero into an element of the array; the starting address `a` of the array is in the immediate field of the instruction. The fourth instruction adds four to register 8. (Why four and not one? Because the elements of the array are words, and so their addresses are four apart from each other.) The fifth instruction branches back to the beginning of the loop. (We'll talk more later about the branch instructions.)

This version of the program works fine for an exercise that you write in MIPS Assembly Language (hereafter called MAL) and try out in SPIM. But a C compiler wouldn't do it that way. The trouble with this translation is that, in general, the compiler *doesn't* know for sure that the array `a` will be at an address below 2^{15}. Instead, the compiler might translate the C code this way:

```
        la   $8, a
        addi $9, $8, 40
loop:   bge  $8, $9, fini
        sw   $0, 0($8)
        addi $8, $8, 4
        b    loop
fini:
```

In this version we are no longer trying to fit the 32-bit address `a` into the 16-bit immediate field of the `sw` instruction. Instead, we load the 32-bit address into register 8. (How does the `la` instruction manage to hold a complete address? We'll get back to that; the short answer is that `la` (Load Address) isn't a real MIPS machine instruction.) We also load the address of the nonexistent element `a[10]` into register 9; when register 8 reaches this value, the loop will be finished. The `sw` instruction has an offset of zero, so nothing is added to register 8, which must contain the actual address.

In this version, register 8 is a *pointer*.

In effect, I've compiled the C program fragment above as if it had been written this way:

```
int *p;
int a[10];

for (p = a; p < &a[10], p++)
    *p = 0;
```

The early C compilers weren't smart enough to translate the first version of the C code this way. Instead they'd actually maintain a variable `i` whose values ranged from 0 to 10, and would multiply that value by four each time the program needed to compute the address of `a[i]`. That's why a lot of old C code is written with pointer variables, like the one just above, even when that makes the code hard to read and understand! Younger C programmers are more likely to trust the compiler and use the more straightforward array-indexing version.

Loading 32-bit values

The system of base registers and offsets works only if we can get a 32-bit address into a register. As mentioned above, MAL provides a Load Address instruction, but there can't be a corresponding machine language instruction because there would be no room for the opcode or the register number. Instead, the assembler translates LA into two real instructions. If you say

```
la   $8,0x12345678
```

the assembler will produce the following two actual machine instructions:

```
lui  $8,0x1234
ori  $8,0x5678
```

The first instruction, Load Upper Immediate, loads its 16-bit immediate operand into the left half of the register, setting the right half to zero. The next instruction, Or Immediate, essentially copies its 16-bit immediate operand into the right half of the register. (We haven't talked about the logical operations yet; for now, just take this on faith. We'll come back to it next week.)

I've used a hexadecimal address in this example to make it easy to see how the address is divided into a left half and a right half. But even if you use a symbol as the address (`la $8,a`) the value of that symbol still has a left half and a right half.

MAL also has a pseudo-instruction LI (Load Immediate) that's just like LA except that its 32-bit operand is data—an integer, for example—rather than a pointer. These two are essentially the same.

Accessing Global Variables

The method of loading a pointer into a register works to access data anywhere in memory — global data, heap, or stack. But there's also a slightly faster way to access global data. By convention, when your program starts running, register `$28` (`$gp`, for "global pointer") contains the address 2^{15} more than the beginning of the global data area. Since I-format offsets are 16-bit signed integers, this means that adding an offset to `$28` can generate any address within the first 2^{16} bytes of the global data area.

Branch and jump instructions

We've looked at how the MIPS gets around the need to cram a 32-bit *data* address into a 32-bit instruction, using a base register and an offset. But some instructions, the ones for jumping around in a program, have *instruction* addresses as operands. For example, the instruction

```
        bge  $8, $11, bigger
```

tells the computer that if the value in register 8 is greater than or equal to the value in register 11, it should get its next instruction from the location named `bigger`. (If not, the computer will continue with the next instruction in sequence.)

How is the address `bigger` represented in the instruction? We can't exactly use a base/offset notation, for two reasons: This instruction already has two register operands, so there is no room in I-format for a third (base) register; also, we don't ordinarily have a processor register containing the address of an instruction.

The secret is that the processor *does* have a register containing the address of the next instruction, but it isn't one of the usual 32 numbered registers, and it doesn't have a dollar-sign name or number because it isn't explicitly used in machine language instructions. This special register is called the Program Counter or PC. Every time the processor needs another instruction, it loads the word from the memory location in the PC into its instruction decoding circuitry, and then adds 4 to the PC to prepare for the next instruction.

So a branch instruction does, in effect, use the base/offset system, but instead of an explicitly named base register, branches use the PC as the implicit base register. The offset can be positive or negative. (Usually positive offsets are used in compiling an `if`, whereas negative offsets are used for looping constructs such as `for`.) One more little wrinkle is that, since the branch offset must be a multiple of four, what is actually stored in the offset field of the instruction is the offset divided by four. This means that the range through which a branch instruction can branch is plus or minus 2^{15} instructions, rather than the 2^{15} *bytes* (which is 2^{13} instructions) that you might expect. So, for example, in the program fragment

```
        la   $8, a
        addi $9, $9, 40
loop:   bge  $8, $9, fini
        sw   $0, 0($8)
        addi $8, $8, 4
        b    loop
fini:
```

that we saw earlier, the value stored in the offset field of the bge instruction is 3. (Remember that when the machine does the branching, the PC already points to the instruction *after* the branch. So an offset of 0 would be the sw instruction, an offset of 1 would be the addi, and so on.)

The MAL unconditional branch instruction b does not correspond to an actual machine language instruction. Rather, the assembler translates it into a conditional branch that always succeeds, such as

```
        beq   $0, $0, loop
```

What if your program has to branch to an instruction more than 2^{15} away? This is a rare situation (2^{15} is 32,768 instructions), but the MIPS includes a Jump (j) instruction whose range is 2^{26} instructions. It uses the J format:

```
+------+--------------------------+
|opcode|          new PC          |
|  6   |            26            |
+------+--------------------------+
```

When the program executes a j instruction, the machine constructs a new PC as follows: The leftmost four bits are the same as in the old PC. The next 26 bits are copied directly from the instruction. And the rightmost two bits are always zero, since an instruction address must be a multiple of four. (What if you want to go even further away? Then you must get the new instruction address into a general register yourself, and then use the jr instruction, for Jump Register, to copy that register's value into the PC.)

The j instruction is always unconditional. If you want a conditional jump to a far-away address, you must use a conditional branch followed by an unconditional jump. That is, if you want to say

```
        jge   $8, $9, foo              # nonexistent instruction
```

what you say instead is

```
        blt   $8, $9, nofoo            # condition of branch is reversed
        j     foo
nofoo:
```

Pointers in C

Here is a possible implementation of strcpy, the C library routine that copies a character string from one place to another:

```
int strcpy(char *out, char *in) {
    int i;
    char c;

    i = 0;
    do {
        c = in[i];
        out[i] = c;
        i++;
```

```
    } while (c != '\0');
    return i-1;
}
```

This implementation should be easy to understand, because it uses array indexing. It copies each element of the character array `in` to the same numbered element of the array `out`, until a null character is seen.

But that's not how `strcpy` is generally written. Instead it's done this way:

```
int strcpy(char *out, char *in) {
    char *p = in;

    while (*out++ = *p++) ;
    return p-in;
}
```

This tricky program exemplifies many of the worst features of C: It uses the fact that the assignment operator `=` returns a value, which is what the `while` tests; it uses the coincidence that a value of zero means both the null character and false; the `while` has no body because its work is done by side effects within the test expression. Nevertheless, you have to know how to interpret a program like this using pointer variables.

We haven't talked about procedure calling yet, so let's translate just the body of the procedure to MAL. I'll allocate two character arrays and use registers for the pointer variables: `out` in `$4`, `in` in `$5`, and `p` in `$8`. We'll put the return value into `$2`.

```
        .data
inbuf:  .asciiz "This is a null-terminated character string"
outbuf: .byte   0:100

        .text
strcpy: la   $4, outbuf             # out = <address of outbuf>
        la   $5, inbuf              # in = <address of inbuf>

        add  $8, $5, $0             # p = in
loop:   lb   $9, 0($8)              # $9 = *p
        addi $8, $8, 1              # p = p+1
        sb   $9, 0($4)              # *out = *p (before increment)
        addi $4, $4, 1              # out = out+1
        bne  $9, $0, loop           # while $9 nonzero
        sub  $2, $8, $5             # return p-in
```

Pointer arithmetic

The `strcpy` example is deceptively simple, because the pointers are `char` pointers, and so the C expression `p++` (or `p = p+1`) translates straightforwardly into

```
        addi $8, $8, 1
```

and the expression `p-in` translates equally straightforwardly into

```
        sub  $2, $8, $5
```

But the meaning of pointer arithmetic isn't so obvious in C if the pointers point to something larger than one byte. Suppose we wrote `intcpy`, a procedure exactly like `strcpy` except that it copies a zero-terminated array of integers instead of an array of characters:

```
int intcpy(int *out, int *in) {
    int *p = in;

    while (*out++ = *p++) ;
```

```
    return p-in;
}
```

Here's the MAL translation (leaving out the initialization of `in` and `out`):

```
        add  $8, $5, $0             # p = in
loop:   lw   $9, 0($8)              # $9 = *p
        addi $8, $8, 4              # p = p+1
        sw   $9, 0($4)              # *out = *p (before increment)
        addi $4, $4, 4              # out = out+1
        bne  $9, $0, loop           # while $9 nonzero
        sub  $2, $8, $5             # return p-in
        div  $2, $2, 4              # (in units of words)
```

One obvious difference is that this program uses `lw` and `sw` instead of `lb` and `sb`, because an integer is a word, not a byte. But what's more interesting is that the C expression `p+1` is now compiled into an instruction that adds *four* to the pointer. In C pointer arithmetic, `p+n` really means

```
p + (n * sizeof(*p))   /* but don't write this in your C program! */
```

Offsets and structs

Suppose you want to build linked lists of integers in C and so you build a structure like this:

```
struct node {
    struct node *next;
    int value;
};
```

You access the fields of the structure with instructions like this one:

```
struct node *p;
int x;

x = p->value;
```

Here's how that looks in MAL:

```
        lw $8, p            # get pointer in register
        lw $9, 4($8)        # get value field
        sw $9, x
```

Here's a case in which both the base register and the offset are useful; the base register has the address of the beginning of the struct, and the offset indicates that the `value` field comes four bytes after the beginning of the struct. If the struct had been declared as

```
struct node {
    struct node *next;
    int wasted[20];
    int value;
};
```

then to find `p->value` we'd have to say

```
        lw $9, 84($8)
```