More I format instructions

Conditional branches

There is a limited number of I format instructions since an I format instruction must be specified entirely by the opcode and the op code field has 6 bits. We have already seen `lw`, `sw`, `beq`. There is also a `bne` (branch if not equal to).

What about other conditional branches? You might expect there to also be a `blt` for "branch if less than", `bgt` for "branch if greater than," `ble` for "branch if less than or equal to", etc. This would make programming easier. However, having so many of these instructions would use up precious opcodes. The MIPS designers decided not to have so many instructions and to take an alternative approach.

To execute a statement such as "branch on less than", you use two instruction. First, you use an R format instruction, `slt` which stands for "set on less than". The `slt` instruction is then combined with `beq` or `bne`. Together, these span all possible inequality comparisons $\leq, <, \geq, >$. Here is an example for the C code:

```
if (i > j)
   i = i + j;
```

We want to branche if `j` is less than `i`. So, we set a temporary variable if this branch condition holds, and then branch if this test variable is not equal to 0. Here is the MIPS code, assuming `i` is `$s3` and `j` is `$s4`.

```
      slt $t0, $s4, $s3
      beq $t0, $zero, Exit    # branch if condition fails
      add $s3, $s3, $s4
Exit:
```

Here is another example. First, the C code:

```
while (i <= j)
   i = i + h;
```

Here we want to exit the loop (branch) if the condition fails, namely, if `i > j`. Here's the MIPS code:

```
MyLoop:     slt     $t0, $s4, $s3
            bne     $zero, $t0, MyLoopExit
            add     $s3, $s3, $s5
            j       MyLoop
MyLoopExit:
```

"Pseudoinstructions" for conditional branches

The instructions `slt`, `beq`, and `bne` can be combined in to give several conditional branches that we would commonly wish to use, such as `blt` (shown above), `ble`, `bgt`, `bge`. As an exercise, see if you can write these "branch if less than or equal to", "branch if greater than", and "branch if greater than or equal to" instructions using `slt`, `beq`, and `bne`. You might find it trickier than it seems. The answer is shown in the slides.

It would be annoying if we had to use the `slt` instruction to write conditional branches in MIPS – our programs would be much more difficult to debug. Fortunately, we don't need to do this. The MIPS assembly language allows you to use instructions `blt`, `ble`, `bgt`, `bge`. These are called *pseudoinstructions* in that they don't have a corresponding single 32 bit machine code representation. Rather, when a MIPS assembler (or a simulator like SPIM) translates the program into machine code, it substitutes two instructions, namely a `slt` instruction and either a `beq` or `bne`.

Heads up: this substitution uses a temporary register to hold the value of the comparison (e.g. `t0` in the example above). So, if your program was using this register for something else, then you will clobber the value that was previously stored there!

One final comment: you may be wondering why the MIPS designers didn't just have separate machine code instructions `blt`, `ble`, `bgt`, `bge`. The answer is that these (I format) instruction would use up precious op codes. The MIPS designers wanted to keep the number of instructions as small as possible (RISC). The cost is that a given program tends to require more of these simpler instructions to have the same expressive power. The benefit is that the RISC architecture allows simpler hardware, which ultimately runs much faster.

Using the immediate field as an integer constant

We have seen R format instructions for adding/subtracting/etc the contents of two registers. But sometimes we would like one of the two operands to be a constant that is given by the programmer. For example, how would we code the following C instruction in MIPS?

$$\texttt{f} \;=\; \texttt{h} \;+ (-10);$$

We use an I-format instruction `addi`,

```
addi    $s0, $s2, -10
```

The "`i`" in `addi` is the same "I" as in "I-format." It stands for "immediate." Recall that I-format instructions have a 16 bit immediate field. The value -10 is put in the immediate field and treated as a twos complement number.

Another example of an R format instruction that has an I format version is `slti` which is used to compare the value in a register to a constant:

```
slti    $t0, $s0, -3
```

In this example, `$t0` is given the value 0x00000001 if the value in `$s0` is less than -3, and `$t0` is given the value 0x00000000 otherwise. Such an instruction could be used in a conditional branch,

```
if (i < -3)
   i = i + j;
```

Signed vs. unsigned instructions

There are various versions of two instructions "add" and "set on less than" :

- `add`, `addi`, `addu`, `addiu`
- `slt`, `slti`, `sltu`, `sltiu`

The `i` stands for immediate and the `u` stands for unsigned. For example, `addu` stands for "add unsigned". This instruction treats the register arguments as if they contain unsigned integers whereas by default `add` treated them as signed integers.

```
addu      $s3, $s2, $s1
```

Note that an adder circuit produces the same result whether the two arguments are signed or unsigned. But as we saw in Exercises 2, Question 11, the conditions in which an overflow error occurs will depend on whether the arguments are signed or unsigned, that is, whether the operation specifies that the registers represent signed or unsigned ints. In this sense, `add` and `addu` do not always produce the same result.

The I instruction `addiu`, treats the immediate argument as a unsigned number from 0 to $2^{16} - 1$. For example,

```
addiu     $s3, $s2, 50000
```

will treat the 16-bit representation of the number 50000 as a positive number, even though bit 15 (MSB) of 50000 is 1. (See "sign extending" below.)

Another example of when it is important to distinguish signed vs. unsigned instructions is when we make a comparison of two values. For example, compare `slt` vs. `sltu`. The result can be quite different, e.g.

```
sltu      $s3, $s2, $s1
```

can give a different result than

```
slt       $s3, $s2, $s1
```

in the case that one of the variables has an MSB of 1 and the other has an MSB of 0.

Sign extending

The 16 bit value that is put in the immediate field typically serves as an argument for an ALU operation (arithmetic or logical), where the other argument is a 32 bit number coming from a register. As such, the 16 bit number must be extended to a 32 bit number before this operation can be performed, since the ALU is expecting two 32 bit arguments. Extending a 16 to 32 bit number is called *sign extending.*

If the 16 bit number is *unsigned* then it is obvious how to extend it. Just append 16 0's to the upper 16 bits. If the 16 bit number is *signed*, however, then this method doesn't work. Instead, we need to copy the most significant bit (bit 15) to the upper 16 bits (bits 16-31).

Why does this work? If the 16 bit number is positive, the most significant bit (MSB) is 0 and so we copy 0s. It trivial to see that this works. If the 16 bit number is negative, the MSB is 1 and so we copy 1s to the higher order bits. Why does this work? Take an example of the number -27 which is 111111111100101 in 16 bit binary. As we can see below, sign extending by copying 1's does the correct thing, in that it gives us a number which, when added to 27, yields 0. Convince yourself that this sign extending method works in general.

```
      1111111111111111 111111111100101      ← -27
+     0000000000000000 000000000011011      ← 27
      0000000000000000 000000000000000      ← 0
```

Manipulating the bits in a register

I format

Suppose we want to put a particular 32 bit pattern into a register, say `0x37b1fa93`. How would we do this? We cannot do this with just one instruction, since each MIPS instruction itself is 32 bits. Instead we use two instructions. The first is a new I format instruction that loads the upper two bytes:

```
lui     $s0, 0x37b1     #load upper immediate
```

This instruction puts the immediate field into the upper two bytes of the register and puts 0's into the lower 16 bits of the register. We then fill the lower 16 bits using:

```
ori     $s0, $s0, 0xfa93
```

If these instructions are put in the opposite order, then we get a different result. The reason is that `lui` puts 0's into the lower 16 bits. Previous values there are lost forever.

[ASIDE: You might think you could use `addiu` instead of `ori` to fill the lower 16 bits. This sometimes works, but surprisingly it sometimes fails. (SPIM generates an error.) Since we are really doing a logical operation here (manipulating bits), it is better to avoid problems and just use `ori`.]

Another bit manipulation that we often want is to shift bits to the left or right. For example, we saw in an earlier lecture that multiplying a number by 2^k shifts left by k bits. (This assumes that we don't run over the 32 bit limit.) We shift left with `sll` which stands for "shift left logical". There is also an instruction `srl` which stands for "shift right logical".

```
sll     $s0, $s1, 7      # shifts left by 7 bits,
                         # filling in 7 lowest order bits with 0's.
srl     $s0, $s0, 8      # shifts right by 8 bits,
                         # filling in 8 highest order bits with 0's.
```

Here is a more complicated example. On the right are shown the resulting values of `$s0`. You should, as an exercise, examine the instructions on the left and ask what is the value of `$s0` after each instruction. The answer is shown on the right.

```
lui     $s0, 0x322b             # $s0 = 0x322b0000
srl     $s0, $s0, 4             # $s0 = 0x0322b000
srl     $s0, $s0, 24            # $s0 = 0x00000003
sll     $s0, $s0, 1             # $s0 = 0x00000006
sll     $s0, $s0, 1             # $s0 = 0x0000000c
sll     $s0, $s0, 1             # $s0 = 0x00000018
```

You might think that this instruction would be I-format. But in fact it is an R-format instruction. You don't ever shift more than 31 bits, so you don't need the 16 bit immediate field to specify the shift. MIPS designers realized that they didn't need to waste two of the 64 opcodes on these instructions. Instead, they specify the instruction with some R format op code plus a particular setting of the funct field. The shift amount is specified by the "shamt" field, and only two of the register fields are used.

MIPS memory (see slides for more details)

At the end of the lecture, I discussed how MIPS memory is partitioned into kernel and user regions, each which is 2^{31} bytes. Each of these is partitioned into a data region and an instruction region. The kernel and user instruction regions are each 2^{28} bytes, or equivalently, 2^{26} words. The instructions are `word aligned`, which means that the starting address of each instruction has 00 in the lower two bits.

Now we see why the jump instruction `j` has a 26 bits field. The jump instruction `j` allows the program to jump to any of the instructions in the current segment. That is, a `j` instruction in the user instruction segment can jump the program to any other instruction in the user segment, and similarly a `j` instruction in the kernel instruction segment can jump to any other instruction in the kernel instruction segment.