Register names (save, temporary, zero)

From what I have said up to now, you will have the impression that you are free to use any of the 32 registers ($0, ..., $31) in any instruction. This is not so, however. Certain registers have restricted use. I will introduce these as needed over the next few lectures.

For the instructions we have seen up to now, one typically uses only registers `$16,..$23`, that is, registers 10***. These registers are named `$s0,..$s7`, where "s" stands for "save".

Another set of registers that you will use commonly is `$8,..$15`, that is, registers 01***. These registers are named `$t0,..$t7` where the "t" stands for temporary. As we will see, these registers typically store values that are needed only temporarily.

As an example, consider the C or Java instruction:

$$\texttt{f} = \texttt{g} + \texttt{h} - \texttt{i};$$

This instruction is implemented by first computing `(g + h)` and then subtracting `i` from the result. The value `g + h` is temporary in the sense that it not stored as part of the instruction. Its value is thrown away after `i` is subtracted. We might want to use a `$t` register for it. Here is an example of the corresponding MIPS instructions:

```
add  $t0, $s1, $s2      # temporary variable $t0 is assigned the value g+h
sub  $s0, $t0, $s3      # f is assigned the value (g+h)-i
```

Another register that gets a special name is `$0`. This register is called `$zero`, and it always contains the value 0. The hardware prevents you from writing into this register!

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).

```
if (a == b)
   f = g + h;
```

The corresponding MIPS instructions might be:

```
            bne $17, $18, Exit1      # if a is not equal to b, goto Exit1
            add $19, $20, $21        # f = g + h
Exit1:                               # Next instruction (following if statement).
```

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" in MIPS, you need two instructions. 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 $\leq, <, \geq, >$. *See (new) Exercises 4.* Here is an example for the C code. I will use C variable names `s1` and `s2` to make it easier to see what's going on.

```
if (s2 > s1)
   x = y + z
```

We want to branch if the condition fails, that is, if `s2` is not greater than `s1`. So, we set a temporary variable to the truth value "`s2 > s1`" or equivalently "`s1 > s2`". We want to branch if the condition is false, that is, if the truth value is 0. Here is the MIPS code, assuming C variables `s1` and `s2` correspond to MIPS registers `$s1` and `$s2` respectively:

```
      slt $t0, $s1, $s2       # set t0 if j < i
      beq $t0, $zero, Exit    # branch if condition fails, that is, $t0 = 0
      add $19, $20, $21       # x = y + z
Exit:
```

Here is another example. First, the C code:

```
while (s1 <= s2)
   s1 = s1 + s5;
```

Here we want to exit the loop (branch) if the condition fails, namely, if `s1 > s2`, *i.e.* if `s2 < s1` . Here's the MIPS code:

```
MyLoop:     slt     $t0, $s2, $s1
            bne     $t0, $zero, MyLoopExit
            add     $s3, $s3, $s5
            j       MyLoop                      # back to top of loop
MyLoopExit:
```

"Pseudoinstructions" for conditional branches

As mentioned above, the instructions `slt`, `beq`, and `bne` can be combined to give several conditional branches that we would commonly wish to use, namely `blt`, `ble`, `bgt`, `bge`. It would be annoying if we had to use the `slt` instruction to write conditional branches in MIPS. 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 in the MIPS. Rather, when a MIPS assembler (or a simulator like MARS) translates the assembly language program that you write into machine code, it substitutes two instructions, namely a `slt` instruction and either a `beq` or `bne`.

Heads up: this substitution uses register `$1` to hold the value of the comparison. You should therefore avoid intentionally using that register when you are programming, because the MIPS assembly may destroy the value you have put in that register! Only use `$t` and `$s` to hold variables.

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 such instructions would use up precious (I format) op codes. The MIPS designers wanted to keep the number of instructions as small as possible (RISC). The benefit of this design is that the RISC architecture allows simpler hardware, which ultimately runs much faster. The cost is that a given program tends to require more of these simpler instructions to have the same expressive power.

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 for a conditional branch.

```
if (i < -3)
   do something
```

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. 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 12, 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 overflow conditions can be tested in the hardware to decide if the program should crash or not. We will talk about such exception handling issues later in the course.

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.

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.
```

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.