By Joshua Cantrell
jjc@cory.berkeley.edu

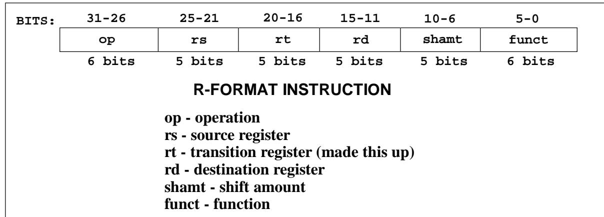
Formats of MIPS Instructions

What are formats?

Formats are patterns that instructions fit so that the microprocessor can tell what to do. Rather than hard coding each possible instruction as a unique possibility[1], the designers of MIPS decided to make instructions follow certain patters that allowed them to extract useful information out of the 32 bits and use it in a more general manner. One group of bits defines one action, another group of bits defines another action.

What are the formats?

There are three types of instruction formats, R-format, I-format, and J-format. The *R* stands for a *register-format instruction*, an *I* stands for an *immediate-format instruction*, and a *J* stands for a *jump-format instruction*. The R-format instructions mainly handle only the register instructions because three of the fields in the format tell the MIPS processor what registers are being used.

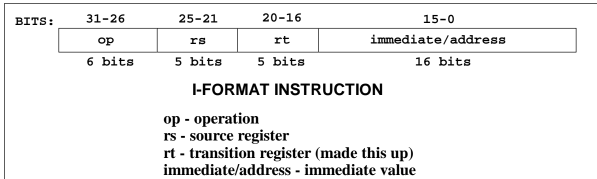


The *operation field* is 6 bits and is found in all of the formats, in the exact same location as shown above. This field tells the microprocessor, not only what type of instruction is being evaluated, but what type of format is being used. In the R-format, the *source* and *transition register fields* tell the microprocessor which registers will be acting as source registers (registers left unchanged by the operation; only their value is used). The *destination register field*, only existing in the R-format, indicates which register will hold the final value of an operation. The *shift amount field* is used in shift operations to indicate how much to shift a given register. Notice that it is 5 bits, so it can shift 32 bits. The *function field*, possibly getting it's name from "math functions", is used in addition to the operation field for determining the type of R-format instruction.

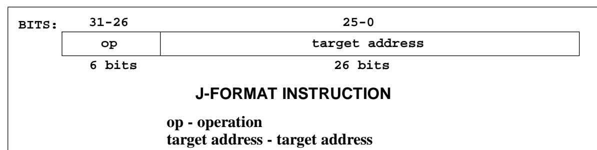
---

[1] This would be 2^{16} possibilites, or 4294967296 different combinations.

By Joshua Cantrell
jjc@cory.berkeley.edu



Whereas many instructions require registers to function, some can take 16 bit values imbedded within the instruction. This is used in MIPS whenever a constant value is encountered. In the I-format, the *transition register field* is used as both a destination register and a source register (hence it's transitional). The *immediate value field* contains a 16 bit value, which may or may not be signed, depending on the instruction. Any value larger than 16 bits must be loaded into a register using *lui* and another instruction, such as *ori*.



The J-format is only capable of operating given a constant value, an address. Some special J-format instructions *do* operate on registers, even though they aren't specified directly[2]. The *target address field* only contains addresses. If an address in excess of a 28 bit value[3] is needed, a register can be used to hold the address.

For a complete listing of the instructions and how they are organized, look in *Computer Organization & Design: The Hardware/Software Interface* by Patterson and Hennessy, on pages A-54 through A-75. A quick reference (although incomplete) can also be found on the back cover of the book.

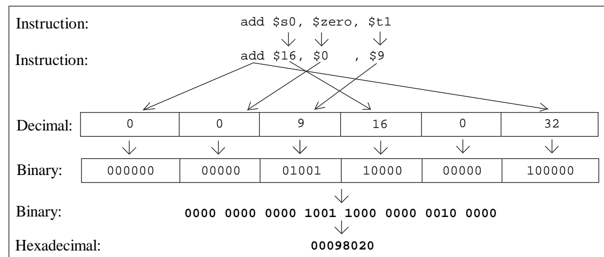
Interpreting the values of formats

Each field in a format consists of a bunch of bits. These bits are read as numbers which define a how the instruction should be interpreted. For example, if you take an R-format instruction, the *rs*, *rt*, and *rd* fields are all 5 bit numbers representing the register being used. (Recall that there are only 32 registers and 5 bits can represent the numbers 0 through 31.) The *op*, *shamt*, and *funct* fields are all treated in the same way. For example what if we had the instruction: `add $s0, $zero, $t1`

---

[2] *jal* is one of these instructions. It loads in the following instruction's address into $ra.
[3] Since moving all of the bits over to the left two places increases the value of the number by 4 times, and instructions are each 4 bytes large, the address is always multiplied by 4 before the operation occurs.

By Joshua Cantrell
jjc@cory.berkeley.edu



Translating instructions from binary into assembly language is just as straight forward. Just reverse the arrows and go backwards.