List of Pseudoinstructions

The following is a list of the standard MIPS instructions that are implemented as pseudoinstructions:

* blt
* bgt
* ble
* neg
* not
* bge
* li
* la
* move
* sge
* sgt

Branch Pseudoinstructions

Branch if less than (blt)

The **blt** instruction compares 2 registers, treating them as signed integers, and takes a branch if one register is less than another.

blt $8, $9, label

translates to

slt $1, $8, $9

bne $1, $0, label

Other Pseudoinstructions

Load Immediate (li)

The **li** pseudo instruction loads an immediate value into a register.

li $8, 0x3BF20

translates to

lui $8, 0x0003

ori $8, $8, 0xBF20

Move (move)

The move pseudo instruction moves the contents of one register into another register.

move $1, $2

translates to

add $1, $2, $0

Load Address (la)

la $a0,address

translates to

lui $at, 4097 (0x1001 → upper 16 bits of $at).

ori $a0,$at,disp

where the immediate (“disp”) is the number of bytes between the first data location (always 0x 1001 0000) and the address of the first byte in the string.

**Pseudo Instructions Introduction (from CS Dept., University of Maryland, College Park)**

### **Introduction I**

Pseudoinstructions means "fake instruction". You might wonder why they exist. When designing a modern ISA, one criteria is to decide whether an instruction should be part of the ISA or not. In the past, if you could see a use for the instruction, it was often added to ther instruction set. This lead to bloated and slow implementations.

In the 1980s, researcher began questioning the assumptions of how to design an ISA. The original thinking was to write ISAs that would be compact and useful to compiler writers. Compactness was important because memory was very expensive up until the 1980's.

Thus, instructions were often variable-sized, i.e., some instructions had as few as one byte, to as many as, say, 8 or more bytes. The thinking was "why use any more bytes than necessary".

Once memory became cheaper, it was reasonable to think that perhaps instruction sizes could be larger, and perhaps fixed in size. This would allow hardware designers to use advanced techniques like pipelining. By having equal sized instructions, it was easier to fetch the instructions, because no decoding was required.

Initially, RISC ISA designers thought that minimizing the total number of instructions was the best way to go. Fewer instructions meant simpler hardware, and simpler hardware meant easier optimization.

Fewer instructions also meant larger volume of code. What used to be executed in one CISC instruction might require ten or more RISC instructions. The hope was that faster hardware (and caches) would offset the increase in code size.

This turned out not to be the case. It was decided that the best way to decide whether an instruction should or should not be included was benchmarking. Use typical C or FORTRAN code, and compile it to assembly. Run the code on a CPU, and see if the time to execute runs significantly faster with the instruction as opposed to without. If it performs better, leave the instruction in. If not, leave it out.

This idea was fairly revolutionary. Instead of designing for compiler writers and memory, the idea was to design ISAs for performance.

However, by taking out some instructions, assembly language programmers would find it a little harder to write code. To make it easier for them, pseudoinstructions were added. Pseudoinstructions do not correspond to real MIPS instructions.

Instead, the assembler, a program that converts assembly language programs to machine code, would then translate pseudoinstructions to real instructions, usually requiring at least one on more instructions.

Pseudoinstructions not only make it easier to program, it can also add clarity to the program, by making the intention of the programmer more clear.

### Pseudo instructions

Here's a list of useful pseudo-instructions.

* **mov $rt, $rs**

Copy contents of register **s** to register **t**, i.e. **R[t] = R[s]**.

* **li $rs, immed**

Load immediate into to register **s**, i.e. **R[s] = immed**. The way this is translated depends on whether **immed** is 16 bits or 32 bits.

* **la $rs, addr**

Load address into to register **s**, i.e. **R[s] = addr**.

* **lw $rt, big($rs)**

Load a word into memory with a 32-bit offset (called **big**). Notice that this is normally not allowed, because only 16-bit offsets are permitted.

Similar pseudo-instructions exist for **sw**, etc.

There are a series of branch instructions that are also useful. They are explained in a later set of notes. Other ISAs often have pseudo-instructions for the convenience of the programmer.

### Translating Some Pseudoinstructions

|  |  |
| --- | --- |
| **Pseudoinstruction** | **Translation** |
| **mov $rt, $rs** | addi $rt, $rs, 0 |
| **li $rs, small** | addi $rt, $rs, small |
| **li $rs, big** | lui $rs, upper( big )  ori $rs, $rs, lower( big ) |
| **la $rs, big** | lui $rs, upper( big )  ori $rs, $rs, lower( big ) |
| **lw $rt, big($rs)** | lui $t0, upper( big )  ori $t0, $t0, lower( big )  add $t0, $rs, $t0  lw $rt, 0($t0) |

where **small** means a quantity that can be represented using 16 bits, and **big** means a 32 bit quantity. **upper( big )** is the upper 16 bits of a 32 bit quantity. The assembler must figure out how to get the upper 16 bits of a 32-bit value (that takes a little work). **lower( big )** is the lower 16 bits of the 32 bit quantity. Again, the assembler must figure out. Thus, **upper( big )** and **lower( big )** are not real instructions.

### **Introduction II**

As you look through the branch instructions, you see **beq** and **bne**, but not **bge** (branch on greater than or equal),**bgt** (branch on greater than), **ble** (branch on less than or equal), **blt** (branch on less than). There are no branch instructions for relational operators!

Why not?

The answer lies in the way the MIPS folks designed the ISA. They decided that instructions could stay in the ISA if there was a noticeable improvement in the speed using benchmarks, otherwise, if it could be written using two or more instructions, without noticeable delay (because they weren't used too often), then those instructions weren't made part of the ISA.

This is a completely different strategy of designing ISAs compared to CISC ISAs. For CISC ISAs, the thinking was to add instructions to make it easier to write compilers. The thought of making the machine run faster was not the first one to come to mind. Only in the early 1980's did some researchers (notably, Hennessy and Patterson) begin to think that complex instructions were not better. Not surprisingly, this coincided with memory becoming cheaper and cheaper. When that happened, processors could be designed differently to take advantage of this economic fact. RISC owes as much of its success to the economics of memory as to any technological advantage it may have over CISC.

Assemblers allow programmers to use pseudo-instructions like **blt**, which it then translates to two or more instructions. Usually, the translations are simple. If you want sophisticated translations, you program in a high-level language.

### slt, slti, sltu, sltiu

To translate these instructions, we'll primarily use **slt** (set on less than). However, there are three other related instructions. Here's a brief summary:

slt $rd, $rs, $rt # R[d] = R[s] < R[t] ? 030::1 : 032

sltu $rd, $rs, $rt # R[d] = R[s] < R[t] ? 030::1 : 032

slt $rt, $rs, immed # R[d] = R[s] < immed ? 030::1 : 032

sltu $rt, $rs, immed # R[d] = R[s] < immed ? 030::1 : 032

Notice I'm using the conditional expression (question-mark/colon operator) in C to explain the semantics.

Here are the differences between all the instructions.

* **slt** performs comparisons of two registers, assuming they store 32-bit 2C representations.
* **sltu** performs comparisons of two registers, assuming they store 32-bit UB representations.
* **slti** performs comparisons of a register and a sign-extended immediate value, assuming the register stores 32-bit 2C representations.
* **sltiu** performs comparisons of a register and a zero-extended immediate value, assuming the register stores 32-bit UB representations.

As you might guess, **slt** and **sltu** are R-type instructions, while **slti** and **sltiu** are I-type instructions.

### Translated Pseudoinstructions

Here's the table for translating pseudoinstructions.

|  |  |
| --- | --- |
| **Pseudoinstruction** | **Translation** |
| **bge $rt, $rs, LABEL** | slt $t0, $rt, $rs beq $t0, $zero, LABEL |
| **bgt $rt, $rs, LABEL** | slt $t0, $rs, $rt bne $t0, $zero, LABEL |
| **ble $rt, $rs, LABEL** | slt $t0, $rs, $rt beq $t0, $zero, LABEL |
| **blt $rt, $rs, LABEL** | slt $t0, $rt, $rs bne $t0, $zero, LABEL |

where we assume **$t0** is not **$rs**, nor **$rt** (if it is, pick another register that's being unused), and **$zero** is **$r0**.

### Machine Code Representation

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **B31-26** | **B25-21** | **B20-16** | **B15-11** | **B10-6** | **B5-0** |
|  | **opcode** | **register s** | **register t** | **register d** | **shift amount** | **function** |
| slt $rd, $rs, $rt | **000 000 (SPECIAL)** | **-** | **-** | **-** | **00000** | **101 010** |
| sltu $rd, $rs, $rt | **000 000 (SPECIAL)** | **-** | **-** | **-** | **00000** | **101 011** |

The dashes are 5-bit encoding of the register number in UB. For example, **$r7** is encoded as 00111.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instruction** | **B31-26** | **B25-21** | **B20-16** | **B15-0** |
|  | **opcode** | **register s** | **register t** | **immediate** |
| slti $rt, $rs, <immed> | **001 010** | **-** | **-** | **-** |
| sltiu $rt, $rs, <immed> | **001 011** | **-** | **-** | **-** |

### Summary

It's useful to know how translate branches for relational operators because they aren't supported directly in the MIPS ISA. Instead, they are pseudo-instructions, which are translated to **slt** and a branch instruction.