What to do when I have a load/store instruction ?

Is there a label involved or a virtual address to compute?

1. A label (such as `ldiq $T1, a;` *`ldq $T0, ($T1);`*):
    a. Find the address the label is pointing at (previously defined in a data declaration structure such as *data{…}*).
    b. Is the address fulfilling the alignment requirement?
        i. Yes
            1. Perform the instruction while respecting:
                a. The little Indian rule (first byte store or load is the least significant one)
                b. Are signed loading instructions involved?
                    i. Yes: Check the sign of the byte, word, longword, quadword loaded and sign-extend it to quadword size.
                    ii. No: If unsigned loading instruction involved, extent the byte, word, longword, quadword, to quadword size by filling the remaining bytes with 0's.
        ii. No
            1. Do nothing
2. A virtual address to compute (*stq $T0, 5($T1)*):
    a. Add the content of the register to the sign-extended 16 bits displacement to obtain the 64 bits address where to load and/or store
    b. Go to 1.b to proceed through the instruction

Loading/Storing and label

To load a quadword from memory (at an address specified by label a) and drop it in register $t0

You need 2 instructions:

1. You need to transfer the address hold by label a (like a move instructions) in a register
    - ldiq $t1, a;
2. Now you can load, in $t0, a quadword at starting memory address hold by the register used in the previous ldiq command (here $t1)
    - ldq $t0, ($t1); or ldq $t0, 0($t1);

To summarize:
Don’t try to do ldq $t0, a; it won't compile

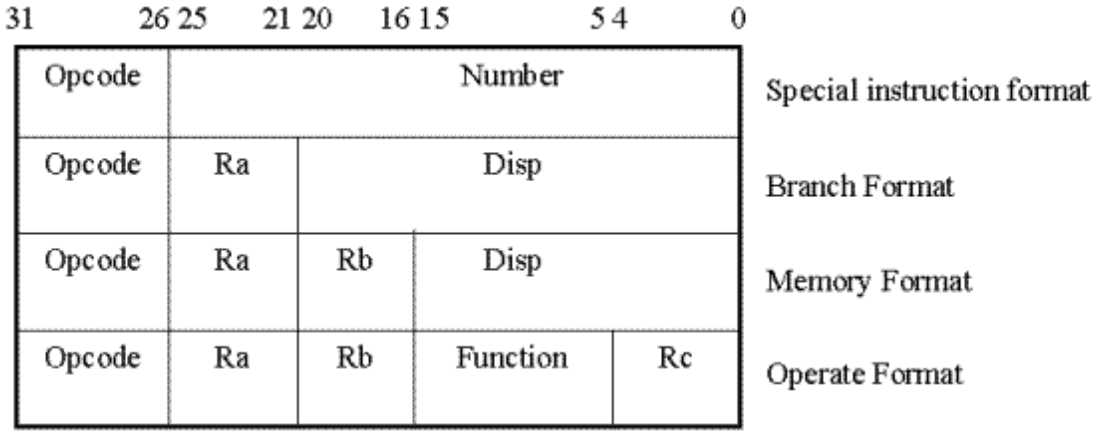
- ldiq $t1, a;
- ldq $t0, ($t1);

will compile.

Warning:
To "store, a quadword holds in register $T0, at memory address specified by label a)" you need to do:

- ldiq $T1, a;
- stq $T0, ($T1);

Branch Instructions



Branch instructions change the control flow of a program.
Two types of branch instructions exist:

- Unconditional branch
- Conditional branch

Conditional branch instructions test register Ra and specify a signed 21-bit PC-relative longword target displacement. Subroutine calls put the return address in register Ra.
Conditional branch instructions can test a register for positive/negative or for zero/nonzero, and they can test integer registers for even/odd. Unconditional branch instructions can write a return address into a register.
There is also a calculated jump instruction that branches to an arbitrary 64-bit address in a register.

Unconditional branch

br is an unconditional branch instruction. The instruction is as below:

br label

It corresponds to:

pc = label;

Label normally consists of one or several letters or digits (always try to use mnemotecnics).

- Labels are used to identify instructions

The instructions are not executed in the order as they appear in the program.

```
start:

    br  next_instruction;
    ldiq$T0, 0x12;
    // some more instructions

next_instruction:

    ldiq$T0, a
    br start ;
```

jmp is the unconditional jump instruction. It has the form:

jmp ($reg);

It is, used to jump to an address when the destination has to be computed at run time. It is often used to implement switch statements.
Essentially it corresponds to:

pc = intReg[reg];

bsr is the branch to subroutine instruction. It has the form:

bsr destination;

It is used to branch to code for a function (subroutine, function, procedure and method are words that mean essentially the same thing). It stores in a register, called the return address register $RA, the address corresponding to the next instruction, just after the *bsr* instruction so that it is possible to return to the right instruction when the subroutine call is over (which is when all the operations included in the subroutine have been performed).
It corresponds to:

intReg[ra] = pc;
pc = destination;

There is a matching instruction to return from a function, the *ret* instruction. It has no operands:

ret;

It restores the pc to its previous value. It corresponds to

pc = intReg[ra];

- We will see details about function invocations later.

Conditional branch

With conditional branch instructions, the machine only jumps to execute the labeled instruction if the condition specified by the instructions is satisfied.

Conditional branch instructions have two operands.
The first operand is a register.
The second operand is a label.
If the value in the register satisfies the condition specified by the instruction (e.g. equal to zero, etc.), then the machine jumps to execute the instruction whose label is the second operand.
Otherwise, the machine executes the instruction, which follows the branch instruction. The format of the conditional branch instruction is as below:

```
    bXY s_reg, label
```

`beq $T0, label`
If $T0 == 0 then jump to label instruction

`bne $T0, label`
If $T0 != 0 then jump to label instruction

`blt $T0, label`
If $T0 < 0 then jump to label instruction

`ble $T0, label`
If $T0 <= 0 then jump to label instruction

`bgt $T0, label`
If $T0 > 0 then jump to label instruction

`bge $T0, label`
If $T0 >= 0 then jump to label instruction

`blbs $T0, label`
If the low bit is set (bit is 1), then jump to label instruction

`blbc $T0, label`
If the low bit is clear (bit is 0), then jump to label instruction

Relational Instructions

A special class of integer operates instruction

`cmpX` instructions are used to test the relation between two values.
The instructions have either two or three operands. The operands can be either registers or a value but the first operand must be a register.
CmpX S_reg1, S_reg2, D_reg
CmpX S_reg1/D_reg, S_reg2

If the test result is true, then 1 is stored in the operand, which holds the result; otherwise, 0 is stored in the operand.

`cmpeq` compares the equality of two values.

```
cmpeq  $T0, $T1, $T3 (cmpeq s_reg1,s_reg2,d_reg)
```
If $T0 == $T1 then $T3 = 1 else $T3 = 0

```
cmpeq  $T0, $T1      (cmpeq d_reg/s_reg1,s_reg2)
```
If $T0 == $T1 then $T0 = 1 else $T0 = 0

```
cmpeq  $T0, 0x2, $T1 (cmpeq s_reg, val, d_reg)
```
If $T0 == 0x2 then $T1 = 1 else $T1 = 0

```
cmpeq  $T0, 0x2      (cmpeq  s_reg/d_reg, val)
```
If $T0 == 0x2 then $T0 = 1 else $T0 = 0

`cmplt` and `cmple` compare two signed quadwords.

```
cmplt $T0, $T1, $T3  (cmplt  s_reg1, s_reg2, d_reg)
```
If $T0 < $T1 then $T3 = 1 else $T3 = 0

```
cmplt $T0, $T1       (cmplt  d_reg/s_reg1, s_reg2)
```
If $T0 < $T1 then $T0 = 1 else $T0 = 0

```
cmplt $T0, 0x2, $T1  (cmplt  s_reg, val, d_reg)
```
If $T0 < 0x2 then $T1 = 1 else $T1 = 0

```
cmplt $T0, 0x2   (cmplt  s_reg/d_reg, val)
```

If $T0 < 0x2 then $T0 = 1 else $T0 = 0

```
cmple $T0, $T1, $T3   (cmple   s_reg1, s_reg2, d_reg)
```
If $T0 <= $T1 then $T3 = 1 else $T3 = 0

```
cmple $T0, $T1    (cmple   d_reg/s_reg1, s_reg2)
```
If $T0 <= $T1 then $T0 = 1 else $T0 = 0

```
cmple $T0, 0x2, $T1   (cmple   s_reg, val, d_reg)
```
If $T0 <= 0x2 then $T1 = 1 else $T1 = 0

```
cmple $T0, 0x2    (cmple   s_reg/d_reg, val)
```
If $T0 <= 0x2 then $T0 = 1 else $T0 = 0

- No >= or > since T0 >=T1 can be done as T1 <= T0
- T0 > T1, it can be performed as T1 < T0

If you want to perform T0 > 0x02, you cannot do *cmplt 0x02, $T0*

- It has to be:
  - ldiq $T1, 0x2;
  - cmplt $T1, $T0 ;

cmpult and *cmpule* compare two unsigned quadwords. The format of *cmpult* and *cmpule* are the same as *cmplt* and *cmple*. They just treat the quadwords as unsigned values and compare them the same way other comparing expressions use.

Example 2: What are the values stored in the relevant T registers when the program terminates.

```
.
.
.
data{
a:   quad 0x123456789abcdef0;
b:   quad 0x9876543210fedcba;
     }
code{
     ldiq          $T0, a;
     ldiq          $T1, b;
     cmpeq         $T0, $T1, $T2;
     ldq           $T3, ($T0);
     ldq           $T4, ($T1);
     cmplt         $T3, $T4, $T5;
     cmpule        $T3, $T4, $T6;
     }
.
.
.
```

Results:

T2=0 (labels a and b point toward different address)
T5 = 0 (since quadword stored in T3 is positive and the one stored in T4 is negative)
T6 = 1 (since this expression deals only with unsigned values and compare them regarding unsigned representation format).

Note:

- Using 2's representation, T3 holds the quadword 0x123456789abcdef0, which is a positive number and T4 holds the quadword 0x9876543210fedcba, which is a negative number equal to –0x6789abcdef012346 (2's representation).
- Using unsigned representation, T3 and T4 holds positive values, lowest one being 0x0..0 and largest one 0xff..ff. Therefore content of T4 is larger than content of T3.