

Low-Level Programming

**Instruction & Instruction Set:**

The language used in computer architecture is called ‘instructions’ in the same way that the language used in the UK is known by the name ‘English’. The vocabulary of ‘instruction’ is called the ‘instruction set’. Since computers cannot understand anything except high and low signals, binary is used. This consists of 1’s and 0’s so that everything is translated into strings of 1’s and 0’s for the computer to be able to understand it.

Groups of bits are named as follows:

* Bit = 0 or 1
* Byte = 8 bits
* half word = 16 bits
* word = 32 bits
* double word = 64 bits

**MIPS r2000:**

A type of language is MIPS r2000, which stands for, Microprocessor without Interlocked Pipeline Stages. It is an assembly language programming type that used 32-bits to operate. Therefore, one instruction in this type of programming will be 32 bits long. There are three main uses of this low-level programming technique and each works in a different way. They are:

* Arithmetic instructions
* Data transfer instructions
* Decision making (conditional branching) instructions

It is important to remember that since this is a low-level programming language and uses only 32-bits to operate, there is only so much that it can do. Therefore the steps have to be simple and follow a unchanging structure.

1. **Arithmetic instructions**

As the name suggests, this instruction set is used for arithmetic calculations, such as, adding a and b to c. The following sum can help us understand how MIPS r2000 works.

In C language: **a = b + c**

In MIPS r2000: **add a, b, c**

Where, add is the operation to be performed and a, b, and c are the operands (the data to be used in the operation). It is to be remembered that in MIPS r2000, the first letter/number, in this case, a, represents where the number has to be assigned. In the operation above, once b and c are added, it can be assigned to a.

However, from this example we can note that only two numbers can be added at a given time. If there are more variables then a different technique can be used.

The sum **a = b + c + d** can be performed by a one line operation in the C language. However, it is not the same for the assembly language in question. It has to be split into two simpler parts for it to be executed:

**a = b + c**

**a = a + d**

so that in both lines, two numbers are being added together at one given time. In both instructions, the answer is assigned to a. In this way, each instruction stays a 32-bit type.

1. **Data transfer instructions**

Again, as the name suggests, data transfer instruction involves the moving of data from one place to another – from the memory to a register. In the above example, **add** and **sub** instructions were used, however, in this case, an **lw** (load word) instruction is used. From the example below we can understand exactly what the lw function does:

In C language: r1 = **mem[100 + r2]**

In MIPS r2000: **lw $r1, 100 ($r2)**

Here, again, the $r1 is the address where the data will be assigned after is has been executed, similar to the ‘a’ in the add function.

100 is the offset and it signifies the distance from to the address stored at $r2 and this information helps us find the actual data needed.

$r2 is the base register. The base register is the memory assigned to store the data that will be flowing in the programming of a program so that the main memory need not be involved all the time. In this line of programming, it holds the address of the data that needs to be moved from the memory to the register.

We now move on to sw, which is store word. In this command, the data from the register is saved to the memory. It is quite similar to lw, just the opposite.

In C language: **mem[100 + r2] = r1**

In MIPS r2000: **sw $r1, 100 ($r2)**

Again, the first part of the line determines where the data will be saved – or more simply, the place where the address is stored. 100 is the constant (distance) and the last bit of the line is the base address.

1. **Decision making (conditional branching) instructions**

Conditional branching instructions allow deciding between 2 alternatives, based on the results of a test. In the example below, it can be seen that there are two types of branching; ‘branch if equal’ or ‘beq’ and ‘branch if not equal’ or ‘bne’. In this type of instruction, two values are compared and if it matches the given value, then the next step can be done, and if not, it keeps going in a loop until the wanted value is given.

**beq $r1, $r2, label**

**bne $r1, $r2 label**

Here, beq and bne are the functions that need to be executed. $r1 and $r2 are the addresses for r1 and r2 which, if, equal (for the first operation), can move to ‘label’ instruction. If not, it moves on to the next instruction and keeps doing so until r1=r2. For the second instruction, it is the opposite. ‘Label’ can be performed if r1 and r2 are **not** equal. If they are, the next instruction is executed until the requirements are met.

However, in these instructions, comparison is not done. It can only tell whether a value is equal to another or not. In the case that there needs to be a comparison, another line of programming is used. For this the slt (set-on-less-than) and slti (set-on-less-than-immediate) instructions are used. For example,

**slt $r0, $r3, $r4**

**If r3 < r4, r0 is set to 1**

**Else r0 is set to 0**

The instructions are clear; until the loop is not broken, it will not move on.

**slti $r0, $r3, 10**

**If r3 < 10, r0 is set to 1**

**Else r0 is set to 0**

Unlike beq and bne, these instructions do not branch off to an address but set a flag that is stored in the first operand.

1. **Programming in MIPS r2000:**

Write a program with the following instruction:

*bne $register1, $register2, label if $register1 is not equal to $register2 then go to label otherwise execute the next instruction which is subtract $register2, $register5 and store in $register3. Under label you have to use add immediate instruction to add £register2 and constant 4 into $register1.*

**bne $r1, $r2, Label** #If $r1 is not equal to $r2 then go to label otherwise move

to next instruction

**sub $r3, $r2, $r5** #$r2 value and $r5 value is subtracted and saved to $r3

**Label: addi $r1, $r2, 4** #$r2 is added to constant 4 and saved to $r1