Introduction to MIPS Instruction Set Architecture

The MIPS used by SPIM is a 32-bit reduced instruction set architecture with 32 integer and 32 floating point registers. Other characteristics are as follows:

1. Addressing modes: LIKE RISC, MIPS is a load-store architecture with only one addressing mode that is **Displacement Mode**. The following two instructions show how these modes are used.

Load /Reading data from memory

**LW R2, 100(R3)**

The ‘**LW**’ mnemonic means Load a Word (four bytes of data) from memory. **R2** is destination to where the data will be loaded. Previous value of **R2** will be over written. In displacement mode the physical address is calculated by adding **R3** and 100 (Value of **R3 + 100**). So if **R3** is 20, the final address will 120 and **R2** will have a copy of 4 bytes of data stored at memory address 120. Note that value of **R3** will remain unchanged. General Format is **LW rt, Imm(rs)**, where **Imm** is a 16-bit value.

Storing/Writing a word to memory

**SW R5, 100(R6)**

The ‘**SW**’ mnemonic means Store a Word (four bytes of data) to memory. **R5** provides the source value to be written to memory at address **R6 +100**. So if **R6 = 45**, this instruction will store the value of **R5** at memory location 145. IMPORTANT: Note that value of **R5** and **R6** remain unchanged, only the memory location at address (100+R6) get a copy of data in **R5**. General Format is **SW rt, Imm(rs)**, where **Imm** is a 16-bit value.

In MIPS this is an immediate type (I-Type) instruction with following format: Note all MIPS instructions are 32-bit (4-bytes) with left most 6-bits as opcode. For LW 6-bit opcode is 100011₂ and for store word it 101011₂. Since there are 32 registers, so 5-bits are used to specify a register field.

2. ALU instructions. These are R-Type instructions that perform mathematical and logical operations. The format of these instruction is as follow:

**ADD R8, R6, R4**

Here the value of **R6** and **R4** are added and result is written in **R8**. After the operation, **R8** is updated with **R6+R4** and previous value of **R8** is destroyed. **R6** and **R4** remain unchanged. The opcode for all R-type instruction is 000000. The general format is OpCode rd, rs, rt, where opcode can be ADD(100000), SUB(100010), OR(100101), NOR(100111), etc. The ALU function is specified by 6 bits on the LSB side as shown below:
Excercise2: Find 32-bit code for OR R7, R18, R12 and SUB R5, R4, R31

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

3. Branch instructions. There are only two branch instructions, BEQ and BNEQ. The format is as follows:

\[
\text{BEQ R5, R7, 25}
\]

This instruction change the value of Program Counter if the values of R5 and R7 are equal. The new value will be PC+4+25*4. Note that each instruction is 4-bytes long, PC is incremented by 4 after ALL instructions. For branch instruction the immediate field represents branching distance in terms of number of instructions, hence multiplication of 4 in the above instruction. General format is BEQ rs, rt, branch offset.

Exercise 3: Find 32-bit code for BNEQ R7, R8, -5. If the PC value for branch instruction is 124 what will be new value of PC if the branch is taken.

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
<td>OPcode, 000100 for BEQ 000101 for BNEQ rs field rt field Branch offset in Number of Instructions</td>
</tr>
</tbody>
</table>

4. I-Type ALU instructions. You can use 16-bit constant for immediate operations. The 16-bit constant is sign-extended in hardware for calculation on 32-bit registers. Example is as follows:

\[
\text{ADDI R4, R5, 500}
\]

This instruction add 500 immediate value to R5 and result is update in R4. General format is OpCode Rt, Rs, Imm. The opcode for ADDI is 001000, ORI is 001101, etc. The format is as follows

Exercise 5: What is the 32-code for ADDI R7, R8, 600? There is no SUBI, why?

<table>
<thead>
<tr>
<th>Bit</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
<td>OPcode, rs field rt field 16-bit Immediate Data</td>
</tr>
</tbody>
</table>

5. Special Instructions. There are some more few special instructions. I will only discuss one over here. That is \textbf{Set on Less than} instruction used for branching. The format is as follows:

\[
\text{SLT R5, R4, R11} \\
\text{SLTI R6, R7, 100}
\]

In the first instruction, the value of R5 = 1 if R4 <R11, 0 otherwise. In second instruction, the value of R6 is 1 if R7 <100. The format is same as R-type for first instruction and I-type for second.

6. Shift Instructions. These are used to logically shift a register. The format is as follows:

\[
\text{SLL R5, R4, 4} \\
\text{SRL R7, R4, 21}
\]
The instruction format is as follows:

| Bit   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Field | 0  | 0  | 0  | 0  | 0  | 0  | rs field = 00000 | rt field | rd field | Shift Amount | Function Field |

7. Unconditional Jump. These are **J-Type** This is used to jump to long distance. The format is simple:

\[
\text{j 200000}
\]

This allows a jump to any PC value. The format is as follows:

| Bit   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Field | 0  | 0  | 0  | 0  | 0  | 1  | 26-bit Jump address |
Beginners write assembly language programs by first writing a code in C and then converting it to assembly language. A reverse process is also often employed by recognizing typical assembly language constructs and converting them to high-level directly.

### C Code

**Integer Declaration**

```c
int count = 10; l, j, k;
```

**MIPS Equivalent**

```mips
count .word 10
i .word
j .word
```

**Byte size variable**

```c
char sum = 1, total= 5;
char uni[10] = “university”;
```

**MIPS Equivalent**

```mips
sum .byte 1
total .byte 5
uni .byte “u”, “n”, “i” etc;
```

**Strings**

```c
string str (“hello world”);
```

**MIPS Equivalent**

```mips
str .asciiz “hello world”
```

**Arrays**

```c
int age[100];
```

**MIPS Equivalent**

```mips
age .word 0:100 ; 100 words initialized to zero
```

**Assignment**

```c
count = count + 1;
```

**MIPS Equivalent**

```mips
la R1, count   # (load address works in spim)
#R1 has address of count
lw R2, 0(R1) # R2 has the value of count
addi R2, R2, 1 # Increment
sw R2, 0(R1) # Save count, if need be
```

```c
sum = age[20] + age[0] + count;
```

**MIPS Equivalent**

```mips
la R3, age  #R3 has base address of age;
lw R4, 80[R3] #20 * 4 = 80, R4 = age[20]
lw R5, 0(R3) #R5 = age[0]
add R4, R4, R5 #
add R4, R4, R2 #R2 had value of count above
# R4 has final sum
```

The following shows a more complex conversion from C to assembly.

### CONTROL

```c
while (count < b)
{
    age[count] = age[count+1];
    count = count+1;
}
```

```mips
.data
age .word 12 23 34 54 23 45

.code
la R4, age ; Load base address of age
addi R2, R0, 0 ; clear R2 to 0.
lable1:
sll R6, R2, 2 ; R6 = R2 * 4
add R3, R6, R4 ;R2= count, R4 = age
:add R3, R6, R4 ;R3 = address of age[count]
lw R7, 4(R3) ;R7 = age[count+1]
sw R7, 0(R6) ;age[count] = R7
addi R2, R2, 1 ; Count = count +1;
slt R8, R2, R10 ; Assume R10 = b
;bnez R8, R0, label1
```
Introduction to SPIM

SPIM is simulator commonly used for simulating MIPS assembly language program. The following points can be noted about SPIM.

1. SPIM simulates 32-bit architecture with 32 integer and 32 floating point register. Although these register are R0 to R31 the SPIM assigns them names that start with $ sign as explained below:

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>R0</td>
</tr>
<tr>
<td>$at</td>
<td>R1</td>
</tr>
<tr>
<td>$v0</td>
<td>R2</td>
</tr>
<tr>
<td>$v1</td>
<td>R3</td>
</tr>
<tr>
<td>$a0</td>
<td>R4</td>
</tr>
<tr>
<td>$a1</td>
<td>R5</td>
</tr>
<tr>
<td>$a2</td>
<td>R6</td>
</tr>
<tr>
<td>$a3</td>
<td>R7</td>
</tr>
<tr>
<td>$t0</td>
<td>R8</td>
</tr>
<tr>
<td>$t1</td>
<td>R9</td>
</tr>
<tr>
<td>$t2</td>
<td>R10</td>
</tr>
<tr>
<td>$t3</td>
<td>R11</td>
</tr>
<tr>
<td>$t4</td>
<td>R12</td>
</tr>
<tr>
<td>$t5</td>
<td>R13</td>
</tr>
<tr>
<td>$t6</td>
<td>R14</td>
</tr>
<tr>
<td>$t7</td>
<td>R15</td>
</tr>
<tr>
<td>$s0</td>
<td>R16</td>
</tr>
<tr>
<td>$s1</td>
<td>R17</td>
</tr>
<tr>
<td>$s2</td>
<td>R18</td>
</tr>
<tr>
<td>$s3</td>
<td>R19</td>
</tr>
<tr>
<td>$s4</td>
<td>R20</td>
</tr>
<tr>
<td>$s5</td>
<td>R21</td>
</tr>
<tr>
<td>$s6</td>
<td>R22</td>
</tr>
<tr>
<td>$s7</td>
<td>R23</td>
</tr>
<tr>
<td>$s8</td>
<td>R24</td>
</tr>
<tr>
<td>$s9</td>
<td>R25</td>
</tr>
<tr>
<td>$s10</td>
<td>R26</td>
</tr>
<tr>
<td>$s11</td>
<td>R27</td>
</tr>
<tr>
<td>$s12</td>
<td>R28</td>
</tr>
<tr>
<td>$s13</td>
<td>R29</td>
</tr>
<tr>
<td>$s14</td>
<td>R30</td>
</tr>
<tr>
<td>$s15</td>
<td>R31</td>
</tr>
</tbody>
</table>

2. Of these $a0-$a3 are used for parameter passing in proc (we not cover that in this course), but can use this registers as temporaries. $t0 to $t9 are temporaries, and so are $s0 to $s7 for this course.

3. For printing (or reading) some values, a special command/procedure called SYSCALL is used that uses $v0 for parameter passing as follows:

<table>
<thead>
<tr>
<th>Service</th>
<th>Value in $V0</th>
<th>Argument</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print an integer</td>
<td>1</td>
<td>$a0 = Number to be printed</td>
<td>Number returned in $V0</td>
</tr>
<tr>
<td>Read an Integer</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Print a Float</td>
<td>2</td>
<td>$f12 is number to be printed</td>
<td>Number returned in $f0</td>
</tr>
<tr>
<td>Read a Float</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Print a String</td>
<td>4</td>
<td>$a0 contains address of null terminated string (see asciiz keyword in string declaration above)</td>
<td></td>
</tr>
<tr>
<td>Read a String</td>
<td>8</td>
<td>$a0 = address of buffer $a1 = length of buffer (see asciiz keyword in string declaration above)</td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Your First Program in Assembly Language On MIPS

Let us start our first example of writing first program in assembly language to add two numbers.

First of all, let us see what will be the code in MIPS and C.

C code will be as follow:

```c
int a=1234; b= 5678
int sum

void main ()
{
    sum = a+b;
    cout <<sum;
}
```

Assembly Code

```assembly
.globl main  #type globl not global
.data
value1: .word 1234
value2: .word 5678
.text   #Code starts here
main:
    la  $a1, value1  # Load address, a1 has the address of value1
    la  $a2, value2  # Load address, a1 has the address of value2
    lw    $t1, 0($a1)  #Value1 in t1
    lw    $t2, 0($a2)  #value2 in t2
    add  $t0, $t1, $t2 # sum in t0
    add  $a0, $t0, $zero # pass parameter to a0 for printing
    add  $v0, $zero, 1 # Syscall 1
    syscall   # Cout the number
    li $v0, 10   # exit or STOP
    syscall
```

Type the above MIPS code in a file using notepad and save it as “sum.s”. Note use quotes to preserve extension.

Now click on PCSPIM icon on your desktop. Now Load your program using file menu. Say yes if asked to initialize registers. Now the screen would look like as shown on the next page.

Notice light green area where the registers are. Red area is in the text segment and it shows your program. Blue shows its equivalent hex code and dark green shows compiled assembly for MIPS in absolute register terms. Yellow is address of each instruction in instruction memory. Single step the program using F10 key and after running some SPIM kernel code, the program will jump to your main: Now notice how register change and as you single step your program.
Exercise 2:
Finding a maximum in an array.

```c
void main ()
{
    int age[] = {15, 23, 14, 25, 60, 10, 9, 11, 60, 40};
    int max = 0;

    for (int i = 0; i < 10; i++)
    {
        if (max < age[i])
            max = age[i];
    }
    cout << max << endl;
    return;
}
```

Run this code and write the hex code for each instruction as you will need it in next lab. Single step this code to see register values change. Also notice how Program Counter (PC) changes.

**Homework:**
Write a program to sort an array using bubble sort. Home work should be submitted as rollno.s in the folder on Indus/common. It will be tested in the next lab.