DT081/4 Computer Architecture 3 Laboratory

Lab 3: MIPS32 Machine Language Instructions

Study the following small assembly program written for the MIPS32 CPU. The first four lines are *directives* that tell how the program should be translated by the assembler. You can ignore them for now. The three lines started by the label **start** when compiled produce actual machine instructions. Use the MIPS32 instruction set reference manual to explain what these instructions mean and what the program does when executed. Also note the use of comments on each line after #. From now on, all the programs that you write for these labs should be properly commented.

```
        .set noreorder      # Avoid reordering instructions
        .text               # Start generating instructions
        .globl start        # The label should be globally known
        .ent start          # The label marks an entry point

start:  addi    $8, $0, 0x1 # Load the value 1
        addi    $9, $0, 0x1 # Load the value 1
        add     $10, $8, $9 # Add the values

        .end start          # Marks the end of the program
```

Using the Lab Software

You can access the MipsIt programming environment referred to in this laboratory exercise from the shortcut on the desktop and in the directory C*:\MipsIt\MipsIt2000\bin* with the name *MipsIt.exe*. It will be convenient for you to create shortcuts for the files *MipsIt.exe* and *Mips.exe* on the desktop.

- Save all your files in a directory of your own, on the F: drive.
- Create a new project file for each new program.
- Save the old project before creating a new.

Task 1

Start the *MipsIt* programming environment. Create a new assembler project. Create a new assembler file, type in the program above and save the file.

Task 2

Translate the program to machine code, by the *Build* command in the *Build* menu, or by pressing the button *F7*. If the assembler finds any syntactical errors, it will tell on which line the first error is located. Correct all errors and rebuild the program.

Task 3

Start the simulator *Mips.exe* in the *bin* directory. Now go back to *Mipsit* and upload the machine language program to the simulator. Use the command *Upload to Simulator* in the *Build* menu. The instructions will be uploaded beginning on the address 0x80020000. Switch to the simulator

Task 4

Click on the RAM box to see the code that has been uploaded. You will see that the built-in disassembler which interprets memory contents as instructions allows you to inspect the uploaded instructions. Compare the disassembled code with the original code in your assembly file. Leave this window open.

Task 5

It is possible to inspect the contents of the registers and the program counter. Simply click on the CPU box to see the current contents of all the registers inside the CPU. Leave this window open

Task 6

You can control the execution of the program using the CPU menu. The simulator will execute a program instruction by instruction when you select the command *step* in the menu list or the step button in the tool bar. Use the button to execute the program step by step, and inspect what happens with the registers and program counter after each step.

Task 7

It is also possible to let the computer execute a program at full speed and stop when the program counter contains a certain address, a *breakpoint*. Set a breakpoint immediately after the last address of the program, by double clicking on the line. Reset the contents of the program counter register back to 0x80020000 by clicking on it inside the CPU window. Then start the program using the *run*, item in the CPU menu or clicking the *run* button on the tool bar. Investigate what happens with the registers and program counter.
What would have happened if there was no breakpoint?

Task 8

Add a branch instruction that will cause an endless loop. Add the instruction **b start** at the end of the program (followed by a **nop** as usual), assemble, and upload it again.
What happens when you start the new program?
How can you stop it?

Task 9

The simulator disassembler can show memory contents as integers, floating point numbers, ASCII text, or as instructions. What is it that really is stored in the memory?
Write down the memory contents at the address 0x80020000 as an integer, as a floating point number, and an ASCII sequence and as an instruction. How is the conversion of the binary number to each of these formats done?
How can the computer know that the stored bit pattern is an instruction?

Task 10

Add an instruction to the program that sets the value of register $8 to 0xff. Execute the instruction using the command *step* in the *Cpu* menu, or by clicking the step button in the tool bar. Investigate what happens to the registers and program counter.
What is the new value of the program counter? Why?
What is the new value of register $8? Why?

Task 11

The instruction above contained the value of the rightmost operand in the instruction code itself. This is called *immediate addressing*. Now consider the instruction:

```
        add     $8, $8, $9
```

If you haven't done it already, download the MIPS instruction set from my web page. Using the details of the add instruction contained in the manual, translate the above instruction to machine code and enter it in memory at the next available address. Verify that your translation was correct with the disassembler. Set register $8 to 1 and $9 to 2 and execute the instruction. Investigate the result.
What does register $8 contain after the execution? Why?
This addressing mode is called *register addressing*.

Task 12

Finally, consider the following instruction:

```
        lw      $8, 0x0($9)
```

Again using the instruction set, translate this instruction to machine code, enter it at the next available address, and execute it. Let register $9 contain the address to a word where you already know the memory contents.
Explain the new value of register $8.
Explain in detail all the parts of the address of the instruction.
This addressing mode is called *base* or *displacement addressing*.

Conclusions

- How can the CPU jump to a symbolic label, such as **beq $0, $0, repeat**?
- Which registers are affected by a branch instruction?
- How many bytes are used to store one instruction?
- Can the CPU execute an assembly instruction?
- How does the CPU know that a bit pattern is an instruction?

Dr. R. Lynch

Based on an original lab by Jan Eric Larsson from the Department of Information Technology. Lund university, Sweden.