Assignment 3

Objectives:

* IEEE floating point number addition and rounding
* Memory addressing modes
* Assembly instructions
* Reading object code (machine level instruction) expressed in hexadecimal and understanding how these instructions are stored in memory
* Writing a C program that corresponds to given assembly program

Submission:

* Submit your document called **Assignment\_3.pdf**, which must include your answers to all of the questions in Assignment 3.
  + Add your full name and student number at the top of the first page of your document **Assignment\_3.pdf**.
* When creating your assignment, first include the question itself and its number then include your answer, keeping the questions in its original numerical order.
* **If you hand-write your answers (as opposed to using a computer application to write them):** When putting your assignment together, do not take photos (no .jpg) of your assignment sheets! Scan them instead! Better quality -> easier to read -> easier to mark!
* Submit your assignment electronically on CourSys

Due:

* Thursday, Feb. 6 at 3pm
* Late assignments will receive a grade of 0, but they will be marked in order to provide the student with feedback.

Marking scheme:

This assignment will be marked as follows:

* + Questions 1, 2 and 5 will be marked for correctness.
  + Questions 3 and 4 will be marked for completeness, i.e., you get marks for completing (answering) the question, but it is up to you to verify the correctness of your answer by looking up the solutions when they are posted.

The amount of marks for each question is indicated as part of the question.

A solution will be posted after the due date.

1. [3 marks] IEEE floating point number addition and rounding

When adding real numbers expressed in scientific notation (base 10), we must first transform them such that they have the same exponent. For example, 3.1416 + 1.0 x 103 must be transformed to 3.1416 + 1000.0. Once the numbers are expressed with the same exponent, we need to align their decimal point, then we can add them (1003.1416 = 1.0031416 x 103).

The same is true when adding IEEE floating point numbers, except that the base we are working with is 2.

Perform the following IEEE floating point number additions following the algorithm described above, i.e., first, transform the IEEE floating point numbers (expressed as hexadecimal numbers) such that they have the same exponents, align their binary points and add them. Express their sum as an IEEE floating point number, then express this IEEE floating point number as a hexadecimal number. Show your work and clearly show the result of rounding, if rounding occurs.

* 1. 0x43E4FC80 + 0x41C52333
  2. 0x43E4FC80 + 0x41C52339
  3. 0x3E2AAAAB + 0x3F555555

where 0.16666667 approximates 0x3E2AAAAB  
 and 0.83333333 approximates 0x3F555555

1. [7 marks] Memory addressing modes

Assume the following values are stored at the indicated memory addresses and registers:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Memory Address** | **Value** |  | **Register** | **Value** |
| **0x230** | **0x23** |  | **%rdi** | **0x230** |
| **0x234** | **0x00** |  | **%rsi** | **0x234** |
| **0x235** | **0x01** |  | **%rcx** | **0x4** |
| **0x23A** | **0xed** |  | **%rax** | **0x1** |
| **0x240** | **0xff** |  |  |  |

Imagine that the operands in the table below are the **Src** (source) operands for some unspecified assembly instructions, fill in the following table with the appropriate answers.

Note: We do not need to know what these assembly instructions are in order to fill the table.

|  |  |  |
| --- | --- | --- |
| **Operand** | **Operand Value**  (expressed in hexadecimal) | **Operand Form**  (Choices are: Immediate, Register or one of the 9 Memory Addressing Modes) |
| **%rsi** | **0x234** | Register |
| **(%rdi)** | **0x23** | Indirect memory addressing mode |
| **$0x23A** | **0x23A** | immediate |
| **0x240** | **0xff** | Absolute memory addressing mode |
| **10(%rdi)** | **0xed** | “Base + displacement” memory addressing mode |
| **560(%rcx,%rax)** | **0x01 (0x235)** | Indexed memory addressing mode |
| **-550(, %rdi, 2)** | **0x00** | Scaled indexed memory addressing mode |
| **0x6(%rdi, %rax, 4)** | **0xff** | Scaled indexed memory addressing mode |

Still using the first table listed above displaying the values stored at various memory addresses and registers, fill in the following table with three different **Src** (source) operands for some unspecified assembly instructions. For each row, this operand must result in the operand **Value** listed and must satisfy the **Operand Form** listed.

|  |  |  |
| --- | --- | --- |
| **Operand** | **Value** | **Operand Form**  (Choices are: Immediate, Register or one of the 9 Memory Addressing Modes) |
| 0x234 | 0x00 | Absolute memory addressing mode |
| **560(,%rax,4)** | 0x00 | Scaled indexed memory addressing mode |
| **(%rdi,%rcx)** | 0x00 | Indexed memory addressing mode |

1. [2 marks] Assembly instructions

**Requirement 1:**

We would like to write assembly code (instruction(s)) that multiplies the value stored in the register **%esi** by **c**, where **c** is a positive integer constant (fits in 32 bits), and stored their product in the register **%eax**, i.e., **%eax <- c \* %esi**.

In the table below, write the assembly code (instruction(s)) that satisfies **Requirement 1** above and the other requirements found in the **Other Requirements** column:

|  |  |
| --- | --- |
| **Other Requirements** | **Assembly Code (instruction(s))** |
| * Using **two** assembly instruction * c is any positive integer constant (you can use $c in your instruction) | IMUL $c,%esi  Movq %esi,%eax |
| * Using **one** assembly instruction * c = 8 | leaq (,%esi,8),%eax |
| * Using **one** assembly instruction * c = 5 | Leaq %esi(,%esi,4),%eax |
| * Using **two** assembly instructions * c = 21 | IMUL $21,%esi  Movq %esi,%eax |

1. [2 marks] Machine level instructions and their memory location

Consider a function called **arith**, defined in a file called **arith.c** and called from the main function found in the file called **main.c.**

This function **arith** performs some arithmetic manipulation on its **three parameters**.

Compiling **main.c** and **arith.c** files, we created an executable called **ar**, then we executed the command:

**objdump –d ar > arith.objdump**

We display the partial content of **arith.objdump** below. The file **arith.objdump** is the disassembled version of the executable file **ar**.

Your task is to fill in its missing parts, which have been underlined:

**0000000000400527 <arith>:**

**400527: 48 8d 04 37 lea (%rdi,%rsi,1),%rax**

**\_\_\_40052b\_\_\_: 48 01 d0 add %rdx,%rax**

**40052e: 48 8d 0c 76 lea (%rsi,%rsi,2),%rcx**

**400532: 48 c1 e1 \_4\_ shl $0x4,%rcx**

**400536: 48 8d 54 0f 04 lea 0x4(%rdi,%rcx,1),%rdx**

**40053b: 48 0f af c2 imul %rdx,%rax**

**\_40053f\_\_\_\_\_: c3 retq**

1. [6 marks] C program versus assembly program

Do the Homework Problem 3.58 at the end of Chapter 3 and include your program called **decode2.c** below. Make sure you satisfy the following requirements:

* + Variables and constants must be descriptively named.
  + Your code must be commented and well spaced such that others (i.e., TA’s) can read your code and understand it.
  + You cannot use the goto statement.
  + You must write your program using C (not C++) and your program must compile on a CSIL computer using the Linux operating system.

Once you have created you program **decode2.c**, generate its assembly code version using the optimization level “g” (–Og) and call it **decode2.s**. Include it below as well without making any modifications to it.

You do not have to electronically submit your program on CourSys. However, your program must be functionally correct (i.e., it must compile, execute properly and solve this problem).