**Score: \_\_\_\_\_**

**MA2 – A Machine Language Machine**

**Activities**

COMP256 – Computing Abstractions

Dickinson College

Spring 2023

Prof. Grant Braught

**Name:**

**Introduction:**

Today’s class introduced the idea of machine language programming as a level of abstraction above microprogramming. Writing microinstructions forced us to pay attention to bits of information that were either irrelevant to the task we were performing (e.g. the memory address when performing an addition) or were implementation details that would be better hidden from us (e.g. the switch positions). Machine language instructions hide those irrelevant implementation details, allowing us to focus on the relevant information (e.g. just the memory address and registers when moving data, or just the registers and the operation when doing addition). The activities below will give you some experience and practice with machine basic language programs. Then next class we’ll expand on that to see how these programs can do more complex operations such as branching and looping.

You will need to use the Knob & Switch Machine language reference for many of the questions in today’s activities, so it is included here for convenience:



The Knob & Switch that can execute machine language programs can be found here:

* <https://dickinson-comp256.github.io/Knob-And-Switch-Computer/machine.html>

🔑 1. Each of the following parts of the K&S Computer play an important role in executing machine language instructions. Explain in a sentence or two the role that each of them plays.

a. The Control Unit

b. The Program Counter (PC)

c. The Instruction Register (IR)

d. The Instruction Interpretation Unit

🔑 2. Use the K&S Machine Language Reference to figure out what each of the following machine language instructions does**. Express what the instruction does using the *convenient shorthand notation* from today’s class (e.g R2 ← R1 + R0 or MM[5] ← R3).** Be sure to use the specific values from the instructions for the registers and memory addresses in the shorthand in your answer.

a. 1000 0010 0 10 10110

b. 1010 0010 00 11 00 10

🔑 3. Use the K&S Machine Language Reference to **give the binary machine language instruction that will accomplish each of the following operations.**

a. Copy the value in memory address 13 into register 3 (i.e. R3 ← MM[13]).

b. Copy the value in register 2 into register 0 (i.e. R0 ← R2).

c. Add the value in register 2 to the value in register 1 and put the result into register 2

(i.e. R2 ← R2 + R1).

🔑 4. Explain in a few sentences of your own words how machine language instructions are a higher-level abstraction that is built on top of microinstructions. As we have been doing with abstractions, be sure to clearly identify the information that is relevant to the user of the abstraction and the information that the abstraction hides from its user.

5. Write a machine language program to perform the following operation:

* Subtract the value in memory address 12 from the value in memory address 10 and put the result in memory address 15.

Fill in the columns in the table below for your program. **For the “Operation or Value” column, use our *shorthand notation* to describe the operation that is to be performed. Then in the “Machine Language Instruction” column, give the binary value of the machine language instruction that performs the operation.**

Be sure to enter your program into the K&S and test that it works. You should also test your program with a few different values at memory locations 10 and 12 to be sure your solution is general. Finally, be sure to include a HALT instruction at the end of your program so that it will stop.

Remember, you can use a Plain Text Editor (e.g. Notepad on windows or TextEdit on Mac) to type your ML instructions into a file and then load them into the K&S. If you are not familiar with how to create a plain text file on your operating system here is a link that might help:

* *Creating a TXT File*
  + <https://help.debounce.io/kb/acceptable-files/txt/>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | **Memory Address** | **Operation or Value** | **Machine Language Instruction** |  |
|  | 00000 (0) |  |  |  |
|  | 00001 (1) |  |  |  |
|  | 00010 (2) |  |  |  |
|  | 00011 (3) |  |  |  |
|  | 00100 (4) |  |  |  |
|  | 00101 (5) |  |  |  |
|  | 00110 (6) |  |  |  |
|  | 00111 (7) |  |  |  |
|  | 01000 (8) |  |  |  |
|  | 01001 (9) |  |  |  |
|  | 01010 (10) | -2984 |  |  |
|  | 01011 (11) |  |  |  |
|  | 01100 (12) | 10482 |  |  |
|  | 01101 (13) |  |  |  |
|  | 01110 (14) |  |  |  |
|  | 01111 (15) |  |  |  |
|  |  |  |  |  |

6. **Optional Extra Practice**: Write a machine language program to perform the following operation:

* Set memory address 13 to be 5 times what it is when the program starts.

Fill in the columns in the table below for your program.

There are a number of ways to solve this problem. For an extra challenge try to solve it using just 5 ML instructions.

Be sure to enter your program into the K&S and test that it works. You should test your program with a few different values at memory address 13 to be sure it is general. Finally, be sure to include a HALT instruction at the end of your program so that it will stop.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | **Memory Address** | **Operation or**  **Value** | **Machine Language Instruction** |  |
|  | 00000 (0) |  |  |  |
|  | 00001 (1) |  |  |  |
|  | 00010 (2) |  |  |  |
|  | 00011 (3) |  |  |  |
|  | 00100 (4) |  |  |  |
|  | 00101 (5) |  |  |  |
|  | 00110 (6) |  |  |  |
|  | 00111 (7) |  |  |  |
|  | 01000 (8) |  |  |  |
|  | 01001 (9) |  |  |  |
|  | 01010 (10) |  |  |  |
|  | 01011 (11) |  |  |  |
|  | 01100 (12) |  |  |  |
|  | 01101 (13) | 100 |  |  |
|  | 01110 (14) |  |  |  |
|  | 01111 (15) |  |  |  |
|  |  |  |  |  |

🏆 7. In the previous assignment (MA1) we learned about the bitwise OR (|) and the bitwise AND (&) operators. These bitwise operators have a number of practical uses. One of the most common is to separate or combine parts of bit strings. This can be particularly useful with things like the RGB color model that we learned about. For this problem imagine we are using a RGB color model with 4 bits per color, so 12 bits total.

a. If we have variables R and G that have just the red and green components of the color as shown below:

R = 0101 0000 0000

G = 0000 0111 0000

The bitwise OR operation (|) provides a way to combine these colors. To see how, complete the computation shown below:

0101 0000 0000

| 0000 0111 0000

b. While the bitwise OR operation gives us a way to combine colors the bitwise AND operation (&) provides a way to separate them. Imagine we have the color C as shown below:

C = 0101 0111 1011

Show how the bitwise AND operation can be used to remove the green component of this color. Hint: Find a value that when bitwise ANDed with C gives a new color where the green component is 0000 and the red and blue components are unchanged. Show your solution as a computation similar to part a but using &.

🏆 8. In many programming languages (e.g. C/C++/Python, but notably not Java) the value 010 (0000….00002) is interpreted as having the logical value FALSE. Conversely, any non-zero value is interpreted to have the logical value TRUE. Thus, in languages like these a statement such as the following would actually make sense:

int x = 7;

if (x):

<code>

In the above snippet the <code> would execute because x is non-zero, which as described above is interpreted as TRUE. Thus in turn the condition of the if is TRUE. However, if x were zero the <code> would not be executed. If you’d like you can play with this code in C using the repl.it:

* <https://repl.it/@braughtg/IntTruthValues>

Try changing the value of x to both zero and non-zero values and running the code.

These interpretations of TRUE and FALSE also extend to the logical operations in these languages (e.g. &&, ||). So, for example:

int x=7

int y=0

if (x || y):

<code1>

if (x && y):

<code2>

a. In the above snippet will <code1> execute? Briefly explain your answer.

b. In the above snippet will <code2> execute? Briefly explain your answer.

c. Which of <code1> and/or <code2> will execute if y=5?

If you’d like you can check your answers using the repl.it that contains the above code in C:

* <https://repl.it/@braughtg/IntLogicalOperations#main.c>

🏆🏆 9. Now, we saw in activities MA1 that the | and & operations in the K&S computer are *bitwise operations*. That is, they compute the AND or the OR of the corresponding bits of the A and B inputs to the ALU. Usually when we are programming we want the logical operations (e.g. || and &&) not the bitwise operations (& and |). This question explores the similarities and differences between the bitwise and logical operators when applied to integer values as explained in the previous question.

a. Assume that R0 and R1 contain two integer values in 8-bit two’s complement representation. The bitwise operation R2 ← R0 | R1 also computes the logical OR (||) of the values in R0 and R1. That is, while R2 will contain the value of R0 | R1, that value will also always be correct for R0 || R1.

Argue in a few sentences that the bitwise OR operation will always compute the logical OR operation when applied to two two’s complement integers values.

b. Unlike with logical OR, given two 8-bit two’s complement integer values in R0 and R1 the bitwise operation R2 ← R0 & R1 will not necessarily compute the logical AND (&&) of the values in R0 and R1. Give a counter example showing that the bitwise AND operation does not always compute the logical AND operation (i.e. give binary values for R0 and R1 and the result of R0 & R1).

Optional: To help me improve and scope these activities for future semesters please consider providing the following feedback.

a. Approximately how much time did you spend on this activity outside of class time?

b. Please comment on any particular challenges you faced in completing this activity.