|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 🔑 **Essential** 🔑 | | | |  | 🏆 **Enhanced** 🏆 | | | |
|  |  |  |  |  |  |  |  |  |
| \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ |  | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ |

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

**17 – Indirect Addressing**

**Activities**

COMP256 – Computing Abstractions

Dickinson College

Spring 2022

Prof. Grant Braught

**Name:**

Thus far we have seen how a number of high-level language features can be implemented in assembly language (and thus via an assembler in machine language). We first saw, basic arithmetic and logic operations. We then saw branching instructions and how those allow us to implement the full range of high-level language control structures (if, if/else, for, while, etc…). In today’s class we extended what we can implement to include arrays through the use of two new addressing modes (immediate label and indirect) to implement references. In today’s activities, you will gain more experience with these addressing modes and how they are used to implement arrays. At the end, you’ll bring things together by writing a program that combines control structures and arrays to perform a useful computation.

**Assembly Instruction Reference:**

A full reference to all of the assembly language instructions that we have seen so far is provided here for your convenience as you work though the activities.

*Arithmetic and Logic Instructions:*



*Data Movement Instructions:*



*Branching Instructions:*



**Addressing Modes:**

We have now seen version of the LOAD and STORE instructions that work in four different addressing modes (direct, immediate, indirect and immediate label). That can be a lot to keep straight. The following questions will review these modes and hopefully help to ensure that you have a good grip on them before moving on.

*Immediate Addressing Mode:*

🔑 1. Immediate addressing mode can be used with LOAD to put a literal value into a register. For example: LOAD R0 #25 will place the value 25 into R0. Note that the # before the final operand indicates that this LOAD instruction is using *immediate addressing mode*.

a. What base 10 value will be in R3 after the following immediate mode instruction:

LOAD R3 #125?

b. Write an immediate mode LOAD instruction that will put the base 10 value 75 into R5.

*Direct Addressing Mode:*

🔑 2. This question uses the contents of memory illustrated at the right. Assume that the code needed to create the labels and set the values has been executed. Direct addressing uses labels (e.g. A, B, X, M) that represent to memory addresses (e.g. A is 80, M is 92).

Direct mode LOAD and STORE instructions use labels to access the values stored at the corresponding memory addresses. For example, the instruction: LOAD R7 A will place the base 10 value 92 into register 7. Note that the use of a label as the final operand indicates that this instruction is using *direct addressing mode*.

a. What base 10 value will be in register 5 after the following direct mode instruction executes: LOAD R5 M?

b. Write a direct mode instruction that will place the value from memory address 88 into register 10.

c. Assuming that R3 contains the value 77, what effect will the following direct mode instruction have: STORE R3 B?

d. Write a direct mode instruction that writes the value from register 9 into memory address 92.

*Immediate Label Addressing Mode:*

🔑 3. This question uses the same memory diagram as question #2 above. Immediate label addressing mode can be used with the LOAD instruction to load the value of a label (i.e. the memory address) into a register. For example, the instruction LOAD R4 #A will load the value 80 into R4 (note: not the value 92! That would be direct mode). Note that the use of a # before a label as the final operand indicates that this instruction uses *immediate label addressing mode*.

a. What base 10 value will be in R6 after the following immediate label mode instruction executes: LOAD R6 #B?

b. Write an immediate label mode instruction that will load the memory address corresponding to the label M into register 3.

*Indirect Addressing Mode:*

🔑 4. This question uses the contents of memory illustrated at the right. Assume that the code needed to create the labels, allocate the space and set the values has been executed.

Indirect addressing mode uses the value in a register as a memory address from which to retrieve a value. For example, consider the following instructions:

LOAD R2 P

LOAD R5 R2

After the execution of these instructions R5 will contain the value 10. The first instruction uses direct addressing to load the value 100 into R2 from memory address P (60). The second instruction then uses the value in R2 (100) as a memory address and loads the value (10) found there into R5.

a. What base 10 value will be loaded into R3 by the following instructions?

LOAD R0 Q

LOAD R3 R0

b. What base 10 value will be loaded into R4 by the following instructions? You can check the meaning of the +12 in the reference table provided at the beginning these activities.

LOAD R0 P

LOAD R4 R0 +12

c. What base 10 value will be loaded into R7 by the following instructions?

LOAD R5 #QDAT

LOAD R7 R5

d. Assuming that R3 holds the base 10 value 56 and R1 holds the base 10 value 100, what will be the effect of the following indirect mode instructions?

i. STORE R1 R3

ii. STORE R3 R1 +8

🔑 5. Assuming statements have been executed that created the memory image shown to the right. Also assume that R0 holds the base 10 value 124. Complete the table below. For each statement, indicate **the name of the addressing mode** that the instruction is using and the base 10 value that will be loaded into R1. By the instruction.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | **Statement** | **Addressing Mode** | **Value in R1** |  |
|  | **LOAD R1 #5** |  |  |  |
|  | **LOAD R1 T** |  |  |  |
|  | **LOAD R1 #T** |  |  |  |
|  | **LOAD R1 TARR** |  |  |  |
|  | **LOAD R1 #TARR** |  |  |  |
|  | **LOAD R1 R0** |  |  |  |
|  | **LOAD R1 R0 +16** |  |  |  |
|  |  |  |  |  |

🏆 🏆 This one is challenging. Hint: Recall that each LOAD instruction moves 4 bytes from memory to a register. So, the value being loaded here would be half in one memory location and half in another in our diagrams.

|  |  |  |
| --- | --- | --- |
| **Statement** | **Addressing Mode** | **Value in R1** |
| **LOAD R1 R0 +2** |  |  |

**Allocating Arrays:**

🔑 6. Consider the following two arrays that might be created by statements in a high-level language like Java:



a. Write the assembly language statements that will create the labels, allocate memory and initialize the above arrays. Do not write statements to set the references yet.

b. Write the assembly language statements that will initialize the reference P. Note: Using your labels from part a, and the addressing modes we know, it is not necessary to know the precise memory addresses where these arrays are stored to write these statements.

**Setting References:**

🔑 7. In a high-level language arrays have two parts, the name of the array and the array data. The name of the array is then a reference to the array data. For example, the Java statement:

int[] E={5, 10, 15, 20, 25, 30, 35};

Will create the array:



The following assembly language will create a label for the array name (E) and will allocate and initialize the space for the array data (at address EDAT).

E .word

EDAT: .word 5 10 15 20 25 30 35

These two type directives will produce the memory image shown at the right. Answer the following questions assuming that memory configuration exists.

a. What value must be stored at address E to make it a reference to the array data?

b. Write assembly language instructions that will set E to be a reference to the array data.

c. Write assembly language instructions that will copy the value of the reference from E into R7.

d. Assuming your instructions from part c works correctly:

i. Write an assembly language instruction that will place the value of E[0] into R8.

ii. Write an assembly language instruction that will place the value of E[5] into R9.

iii. Write an assembly language instruction that will place the value in R5 into E[0].

iv. Write an assembly language instruction that will place the value in R5 into E[3].

v. Write a sequence of assembly language instructions that will copy E[1] into E[6].

**Array Processing:**

You have now worked with all the pieces necessary to understand how arrays in high-level languages can be implemented in assembly language: Creating labels, allocating the arrays, setting up references and accessing array elements. Once all these operations are in assembly language they can then be assembled into machine language and run on our machine. This section looks at the implementation of few more complex array processing tasks.

🔑 8. Imagine we want to add up all of the values in the array A below. Given what we know so far, we might write something like the following:

A: .word

ADAT: .word 1 2 3 4 5

LOAD R0 #ADAT

STORE R0 A

LOAD R2 R0 \* get A[0] / use R2 for the total

LOAD R1 R0 +4 \* get A[1]

ADD R2 R2 R1 \* add A[1] to the total

LOAD R1 R0 +8 \* get A[2]

ADD R2 R2 R1 \* add A[2] to the total

\* ... Add Code Here ...

STORE R2 STDOUT

HALT

a. Briefly explain why the “+4” and “+8” provide access to A[1] and A[2], respectively.

b. Extend the above program using the same approach so that A[3] and A[4] are also added to R2.

🔑 9. The approach taken in question 8 clearly does not scale very well to larger arrays and doesn’t work at all if we don’t know the size of the array in advance. This question begins to hint at a better way. Instead of using offsets (e.g. +4, +8, …) we can change the value in the register we are using for the indirect address. Assume we have the following code:

A: .word

ADAT: .word 1 2 3 4 5

LOAD R0 #ADAT

STORE R0 A

LOAD R2 #0

LOAD R1 R0

ADD R2 R2 R1



a. Assuming the memory image at the right, what value will be in the following registers after the above statements execute?

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Register** | **Value** |  |
|  | **R0** |  |  |
|  | **R1** |  |  |
|  | **R2** |  |  |
|  |  |  |  |

b. Given your answer to part a, what values would be in the registers after the following statements execute?

ADD R0 R0 #4

LOAD R1 R0

ADD R2 R2 R1

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Register** | **Value** |  |
|  | **R0** |  |  |
|  | **R1** |  |  |
|  | **R2** |  |  |
|  |  |  |  |

c. Given your answer to parts a-b, write three more statements that add the next element of A (i.e. A[2]) to the total in R2.

🏆 10. Notice that the statements you wrote in #9c are the same as those that appeared in #9b. If they weren’t, go back and double check your work and understand why they should be. Those statements should start to make it apparent how you could implement a loop that can traverse an array. Using the idea of adding to the indirect address (e.g. like we did to R0 above), and the structure of a for loop from last class, write an assembly language program that is equivalent to the following high-level language program:

int[] a = {1, 2, 3, 4, 5};

int sum = 0;

for (int i=0; i<5; i++) {

sum = sum + a[i];

}

Be sure to assemble and run your program to be sure that it works.

Hint: The three statements question 9b (or c) will make up the body of the loop. But note, the program will be easier to write in assembly language if you rearrange them a little.

Give your assembly language program here as your answer.

**Why Bother with References?**



You may have noticed that while we created and setup references in the above activities, they really were not strictly necessary. For all of our examples we could have just used the “*DAT*” label to get the address of the 0th element of the array (e.g. LOAD R0 #ADAT), instead of using the reference (e.g. LOAD R0 A).

However, there is some high-level language functionality that we will not be able to translate to assembly language unless we use references. The following questions explore one relatively simple example.

🏆 11. Consider the memory image that is shown to the right. With this memory image, consider the following high level language program:

Read A

if A == 0:

P = Q;

Print P[0];

Will this snippet print 3 or 10? Explain your answer. (Hint: It’s a trick question).

From thinking about #11, it should be clear that in order to know what will be printed by the final line it will be necessary to use the reference in P. That reference will either point to the PDAT or to the QDAT depending upon what the input was. Now you may be able to think of some clever ways to write assembly that does the same thing as this without using the reference in P (e.g. building and if/else structure in assembly instead of the if). However, it would be a lot to ask of a compiler to come up with that translation.

🏆 12. Write an assembly language program that creates the arrays and references shown in the above memory image. Then directly translate the computation shown in the HLL code snippet from question #11 into assembly language. Be sure to assemble and test your program with inputs that cause 3 to be printed and cause 10 to be printed.

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.