Tutorial Sheet 12 – Effective Address, Address Arithmetic, and Arrays (in Sigma16)

Setup

Pen and paper + computer-based exercises

This lab contains some review questions, on translation between high level and low level code, and translation to assembly language, as well as effective addresses and pointers. A model solution will be posted on Moodle during the week after this tutorial is shared. The problem about Program Strange is challenging and you may find it surprising. At the outset, view it as an entertaining puzzle, or perhaps as a mystery story—can you figure it out? Try to solve the puzzle on paper, and then run the program on the computer and see if you were right. Although the program is strange, it is demonstrates an important idea. Understanding this program is a real milestone.

You should try to ensure you read the model solutions for these tutorial/lab exercises, even if you solve all the problems correctly yourself. The tutorial, your efforts in solving them, your own solutions, and then the "model" solutions, are all meant together to give you a positive learning experience. That, at least, is the theory!

Review Problems

1. Explain how the effective address is calculated in an RX instruction. Give an example where the effective address can be used to (a) load a constant into a register; (b) load a variable into a register; (c) load an element of an array into a register.

2. Find the effective address in each of the following instructions. Assume that R2 contains 28, R3 contains 5, and the address of x is $00a8. Assume that four-digit numbers are hexadecimal, and shorter numbers (e.g. 28) are decimal.

   - `load R6,30[R2]`
   - `load R7,40[R0]`
   - `load R8,x[R3]`

3. Explain the difference between the `load` and `lea` instructions.

4. Explain what each of the following instructions does. Use the notation mem[a] to refer to the contents of memory at address a.

```
      load   R3,7[R0]
      lea    R4,8[R0]
      load   R5,nine[R0]
      load   R6,x[R0]
      lea    R7,x[R0]
      trap   R0,R0,R0
x     data   123
nine  data   9
```

5. Explain the purpose of the following registers in the Sigma16 architecture: ir, pc, adr.

Study as an example program: ArrayMax

1. Open the example program named ArrayMax. In the Editor tab, click Open, then Examples, the Arrays, then select ArrayMax.asm.txt and open it.

2. Study the program, including the comments. Pay particular attention to the way the array element `x[i]` is accessed.

3. Assemble the program and step through its execution.

Writing a complete program: DotProduct

The primary aims of this problem are to learn how to write an array traversal, and to practice writing a complete correct program using a systematic development method. These are key skills that you will need throughout your career in computing, as well as in the assessed exercise.

Please follow the systematic programming approach covered in lectures, in the examples you have been given, and described in this handout. In particular, start with the high level algorithm, translate it to the "goto form", then translate that to assembly language, and put in all the comments as in the examples. ***Don't start by writing the code and then adding the comments later.***

Why are we so particular about developing the program this way? After all, you might be able to write a small program like this by just hacking out the instructions randomly. But we are teaching you programming techniques that *scale up to big problems*, and this exercise shows you how to use the techniques on a small program. If you take a random hacking approach, you'll be able to solve very simple problems, but that doesn't mean that you can program. A competent programmer must be able to solve problems that are large and complex.

It's tempting to skip the systematic methodology and just write down the instructions. But experience has shown time after time that people who try to save time by skipping the systematic approach end up spending *more* time on their programs.

A *vector* is just an ordinary array. We are given two vectors of size n, $X = x_0, x_1, \ldots, x_{n-1}$ and $Y = y_0, y_1, \ldots, y_{n-1}$. The *dot product* (also called *inner product*) of X and Y is the sum

$$\sum_{i=0}^{n-1} x_i \cdot y_i = x_0 \cdot y_0 + x_1 \cdot y_1 + \cdots + x_{n-1} \cdot y_{n-1}.$$

We are given the following data values: $n = 3$, the elements of X are $2, 5, 3$ and the elements of Y are $6, 2, 4$. (However, the program must work correctly for different values of n, X, and Y.) Write a program that sets a variable p to the dot product of X and Y. For the given inputs, the result should be $p = 2 \cdot 6 + 5 \cdot 2 + 3 \cdot 4 = 34$.

Develop your program using the following steps:

1. Write initial full line comments giving the name of the program (DotProduct) and the author (you). These go at the beginning of the program.

2. Write the algorithm using high level language notation. Make the algorithm clear and readable, but you don't need to worry about syntax details. This should be included in the program, as full line comments. You can use either a for loop or a while loop.

3. Translate the algorithm into low level language notation. This means that the program consists of assignment statements, go statements, and conditional goto statements (e.g. if $q < r$ then goto loop). This too should be included in the program as full line comments.

4. Translate the low level language program into assembly language. Make a copy of each statement in the low level language program, and insert the assembly language instructions right after it. This means that each statement in the low level program will appear twice: once in the complete listing of the program, and again as a header before the instructions.

5. Define the variables using `data` statements. Use the initial values specified in the problem description.

6. Assemble the program. Find out the address of the variable where the final result will be stored.

7. Run the program. It's a good idea to step through it one instruction at a time, and check the changes to the registers and memory locations (these changes will be highlighted in red). Check that the final result is correct: look in the memory in the address of `p`. Note that the emulator shows all registers and memory locations in hexadecimal.

8. The best way to debug a small program is to step through, one instruction at a time, and to compare your prediction for what should happen with what actually does happen. Don't just aimlessly run the program to completion, making random changes!

9. **Important!** To check that the program has worked correctly, you need to find the value of the result (the variable `p`) when the program halts. To do this, find the address of `p` and look in the memory display.