CSE 30321 – Computer Architecture – Fall 2021

Lab 01: Procedure Calls in ARM Assembly – Programming and Performance
Assigned: September 9, 2021
Due: September 28, 2021 – submit code by 11:59 pm
Suggestion: Start this lab early – not a few days before the lab is due!
Total Points: 100 points

1. Goals

Looking back to the course goals, this lab is directly related to Goal #4 – i.e., being able to "explain how code written in high-level languages like C, Java, C++, Fortran, etc. can be executed on different microprocessors (e.g., Intel chips, ARM chips, etc.) to produce the result intended by the programmer."

2. Introduction and Overview

Broadly speaking, in this lab, you will need to write ARM assembly code to:

A. Build a doubly linked list –discussed in Fundamentals/Data Structures and reviewed below
B. Perform a **InsertionSort** on the list
C. Traverse the list and delete duplicate data points

Below, I describe how to approach each of the above tasks with respect to how to write your ARM assembly code, how to test it with an ARM simulator, etc. We will reference the lab 01 skeleton – `insertion_sort.s` – which can be found in the `Labs` directory in Sakai (see sub-directory `01`).

Step A:

For **Step A**, initial data to be sorted can be found in line 5 of `insertion_sort.s`. (There is a list of 10 items defined in the array `data_to_sort`.) You will need to load each array element and insert it into a doubly-linked list structure that you will build. Below, I describe how to go about this.

- To determine the memory address assigned by the simulator for the first array element (i.e., `data_to_sort[0]`), use the instruction syntax `ldr r3,=data_to_sort`. Again, see line 8 of `insertion_sort.s` for a representative example – which loads the ***address*** of `data_to_sort[0]` (i.e., the ***address*** of data word 34, not the data '34') into r3.

- Data elements will be stored in contiguous memory locations, and subsequent array elements (i.e., `data_to_sort[1]`), can be referenced using instruction syntax discussed in class. For example, if we want to load the value of `data_to_sort[1]` into register r2, and the address of `data_to_sort[0]` is in register r1, you can simply use a load instruction such as `ldr r2,[r1,4]`. (Of course, there are other ways to do this as well.)

- You should iterate through the data array provided in the skeleton to build a *doubly-linked list*.

- Prior to starting your "load loop", i.e., that will load data from memory and insert it into the list, you should execute the add instruction below:
    - The instruction **add r5, r3, 400** is used to "randomly" specify the address for the element at the head of linked list (LL) – e.g., 400 away from the start of the array

- The linked list will be comprised of a 3-element structure – namely **(i)** the address of the previous element, **(ii)** the data word associated with the element, and **(iii)** the address of the next element.

- When adding an element to the end of a list during the build phase, you must call an *insert* function

- Prior to this function call, you should create the address of the next element.
  - List items may not necessarily be contiguous in memory; the **add r6, r6, #32** instruction is included to mimic the fact that the address of the last item in the first list element may not be contiguous with the first item in the second list element
  - You can change the register number (r6) in the instruction above if you desire

- You can then call the *insert* function where **(i)** the address of the current element, **(ii)** the address of the next element, and **(iii)** the data associated with the next element are arguments; whether you choose to/need to return anything is your design decision. To simplify things, you **do not** need to follow the procedure call conventions (e.g., with respect to register assignments) we have discussed in class.

- *Do not* just sit down and start writing ARM assembly code. Instead, think about what your code needs to do. I might suggest that you draw a picture first that indicates what must be done with pointers when a list is traversed, an item is inserted, etc.

  As an example, below I have included a sketch that suggests the design process that I think you should follow – and how you might derive assembly from it. (This example assumes that we want to insert an item at the end of a singly linked list.)

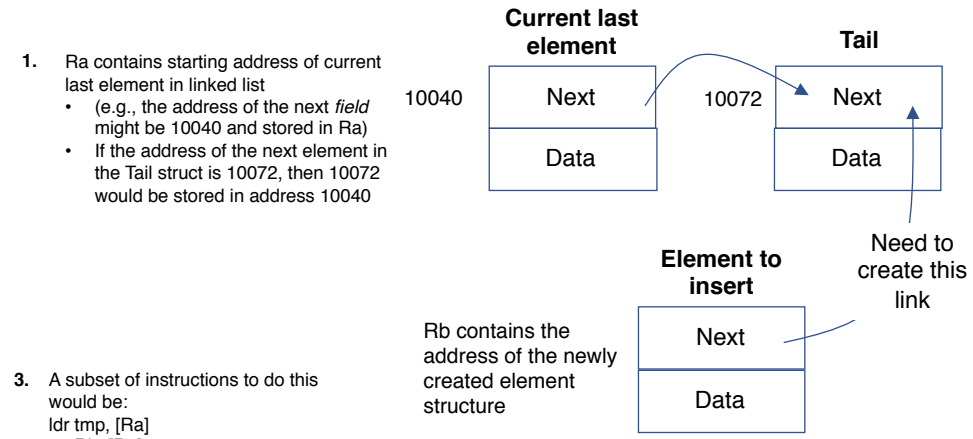

Figure 1: Conceptual view of how to insert element into linked list.

- When you are creating/building a struct/list item, you should assume that entries within the struct are contiguous
  - Therefore, if r1 is the address of the first item of the struct, the code below might be used to update the back pointer with the content of register r9, and the data with the content of register r10.

```
str r9, [r1, 0]      ; previous/back pointer
str r10, [r1, 4]     ; data
```

- The number of list elements is also stored in a 1 element array and can be loaded into a register.
  - See line 6 of `insertion_sort.s`
  - Therefore if the number of items to sort changes we can simply update this number to determine the number of load loop iterations required.

Step B:

For **Step B**, you should perform an **insertion sort** on the items in the list. If you are not familiar with insertion sort, some simple (array-based) pseudo-code is provided below. (You might also do a Google search to find alternative pseudocode if you so choose, look at animations on the Wikipedia page, etc.)

```
void insertionSort(int arr[], int length) {
        int i, j, tmp;
        for (i = 1; i < length; i++) {
                j = i;
                while (j > 0 && arr[j - 1] > arr[j]) {
                        tmp = arr[j];
                        arr[j] = arr[j - 1];
                        arr[j - 1] = tmp;
                        j--;
                }
        }
}
```

When a swap is required (per the code above), you should call a *SWAP* function

- You could just pass either the address of item N-1 or item N into the *SWAP* function; how you choose to do this is your design decision. (Note that you can get other needed addresses via back/next pointers.)
- The swap should occur via pointer manipulation – i.e., move pointers, not data!
- Be sure to pay careful attention to the order of items in the list structure – **previous pointer, data, and next pointer** – and that you are consistent with this ordering throughout your code.

Below, I provide both graphical (Figure 2) and numerical (Figure 3) examples of how pointer manipulation should occur. I assume that the elements of a struct are comprised of **(i)** a back pointer (labeled "previous" in Figure 2), **(ii)** the data in the struct, and **(iii)** the next pointer. In Figure 3, I assume that t*he next pointer in a given struct points to the address of the first entry in the next struct.*

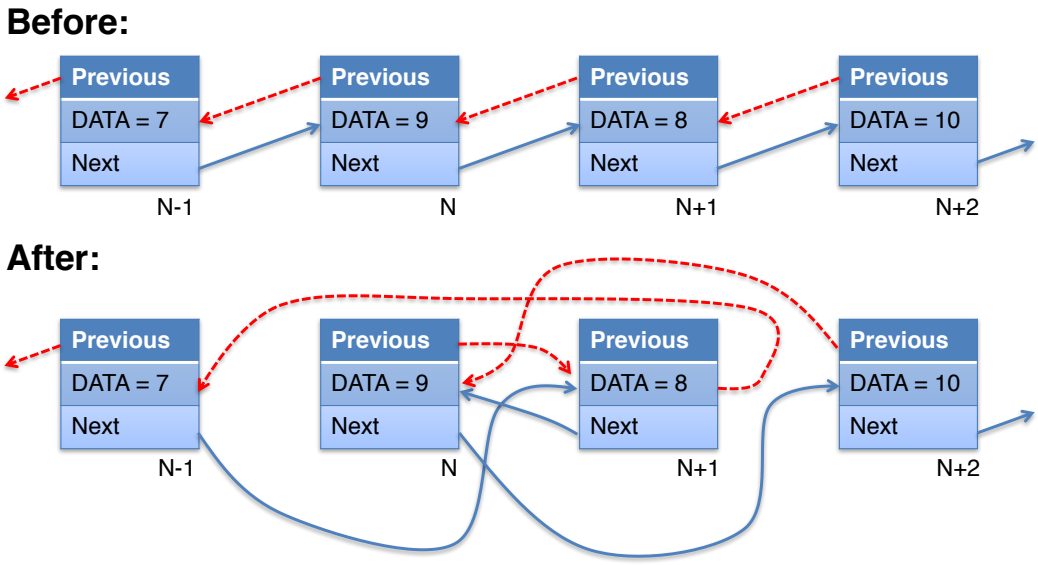

Figure 2: Graphical example of pointer manipulation for *SWAP; do not pay attention to what specific struct element arrows are pointing to – they are just meant to show how one list item is initially mapped to the next, and then remapped in sorted order.*

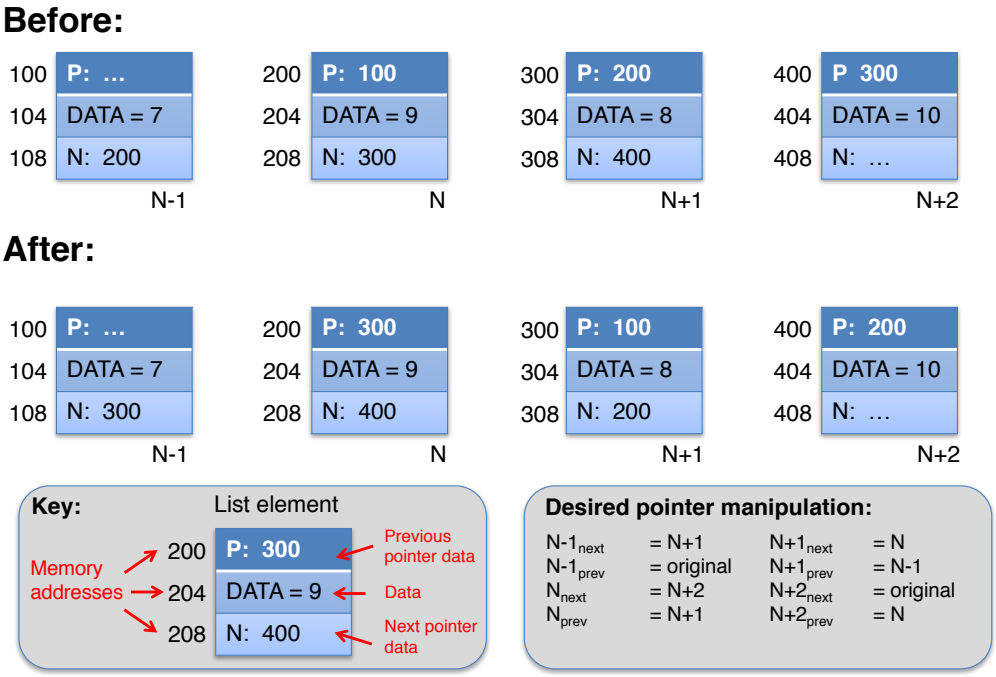

Figure 3: Numerical example of pointer manipulation for *SWAP.*

Step C:

For **Step C**, you should traverse the sorted list and delete all duplicate data entries:

- You can move through the list until you reach a NULL pointer
- (It might be useful to dedicate a register to store the address of the item at the head of the list.)
- When you find a duplicated item, you should call a *DELETE* function to remove the second item. More specifically:
  - Assume the item N-1, item N, and item N+1 hold data elements 7, 7, and 8 respectively
  - In this case, the outcome of the *DELETE* function should be that the *next* pointer of item N-1 points to item N+1, while the *back* pointer of item N+1 points to item N-1

It is OK to add sentinels if needed – i.e., to denote one end of a list – as we will not test your code with negative numbers.

For traversing the list, assume you want to compare data in 1 item to data in the previous item

- Furthermore, assume the address of the current item of the list you are traversing is contained in Rx

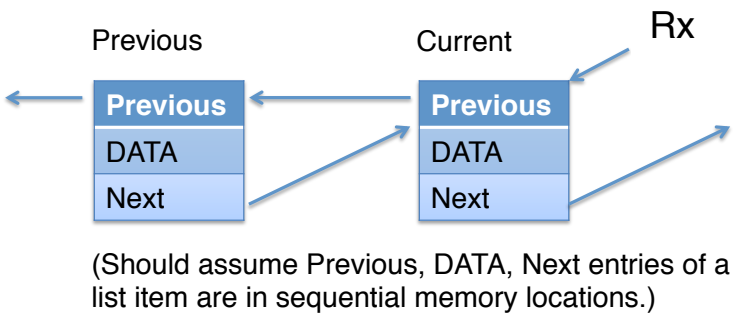

Figure 4: Conceptual view of how to traverse a linked list.

- To get data associated with the current item and the previous item, you might use the following instructions

```
ldr r9, [rX, 4]      ;   [rX, 4] is the address that contains the current item's data;
                     ;   this data will be loaded into r9
ldr r10, [rX]        ;   [rX] is the address of the previous item in the list;
                     ;   this address will be loaded into r10
ldr r11, [r10, 4]    ;   [r10, 4]  is the address that contains the previous item's data;
                     ;   this data will be loaded into r11
```

As of right now, there are no duplicate entries in the initial array – but you can change this to test your code.

3. Utilities

To help you design, debug, and test your code (or other ARM code), you can use VisUAL – a visual ARM emulator.

3.1 Overview

VisUAL is software that will help simulate the execution of ARM assembly programs. It does a context and syntax check while loading an assembly program. It updates register and memory content as each instruction is executed, and can also be used to extract cycles counts from your program, etc. Below, is a brief tutorial on how to use VisUAL.

- Download the source from:
  - https://salmanarif.bitbucket.io/visual/downloads.html
- Note that installation instructions for Windows, Linux, and MacOS are all available.
  - (I have installed and tested the Windows version with no issues.)
  - (I have also installed and tested VisUAL on Mac OS 10.10 (Yosemite), and confirmed that it also works on MacOS 10.12 (Sierra), MacOS 10.13 (High Sierra), MacOS 10.14 (Mojave), etc.)
  - In all instances, simply following the instructions on the website led to a successful installation. However, if you do encounter problems, please let me know.
- A user guide can be found at:
  - https://salmanarif.bitbucket.io/visual/user_guide/index.html

- A window will open as shown in Figure 5.

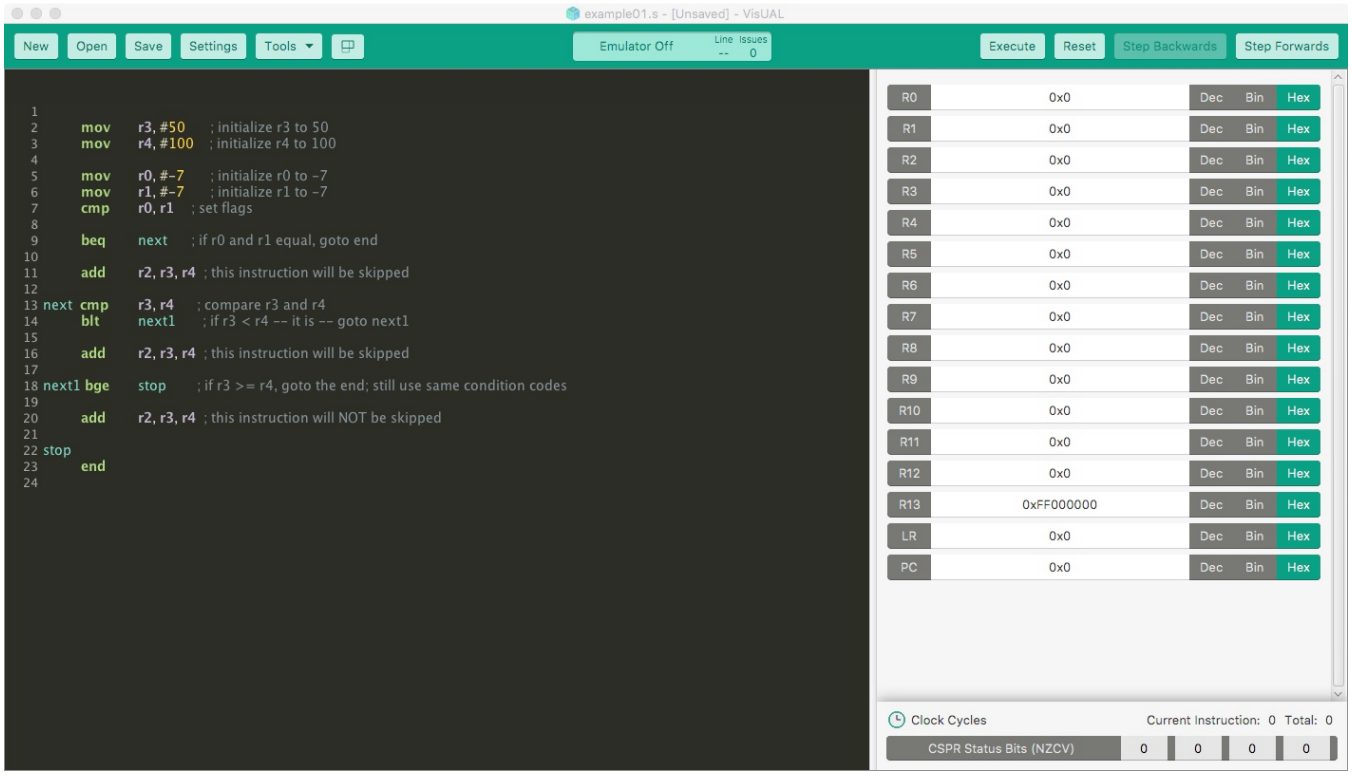

Figure 5: VisUAL environment – ARM assembly is shown at left, register state is shown at right.

- The *ARM assembly code* in shown at left.
    - As you step through instructions, the current instruction will be highlighted
    - The simulator also provides visual information regarding the target of a given branch, the source of a load or store instruction, etc. (Several examples appear below.)

- The *register state* is shown at right. Note that for the ARM ISA assumed with this simulator, there are 16 registers, rather than 32. (Examples of how to extract more information about your program regarding register state are also illustrated below.)
    - R14 maps to the link register (lr) – i.e., to store the return address of a function call
    - R15 maps to the program counter (pc)
    - Note that this simulator DOES NOT support the br instruction
        - To copy the value from the lr (R14) to the PC (R15), use a mov instruction – i.e., mov R15, R14
    - There is no multiply support – use lsl to multiply an array index by 4 if needed.
    - Note that there is no dedicated R0 mapped to the value 0.
    - The status flags are also shown in the *RegisterView* window.
    - Note that you CAN use instructions such as b.le, b.ge, etc. However, the syntax for this is "bge", NOT "b.ge" (as an example)

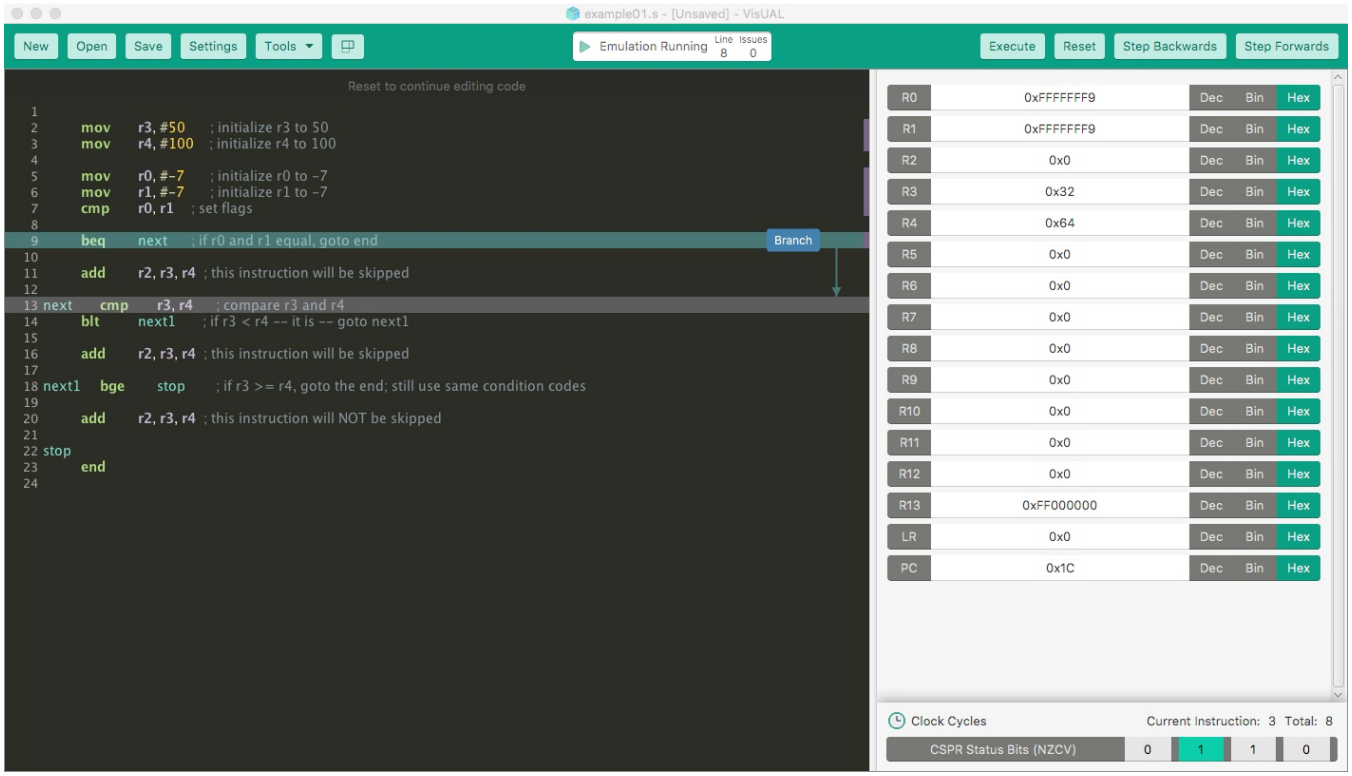

Figure 6: The beq instruction highlighted in this figure will be taken as r0 is equal to r1. (Both registers have the number -7.) Note that the target of the branch is also highlighted.

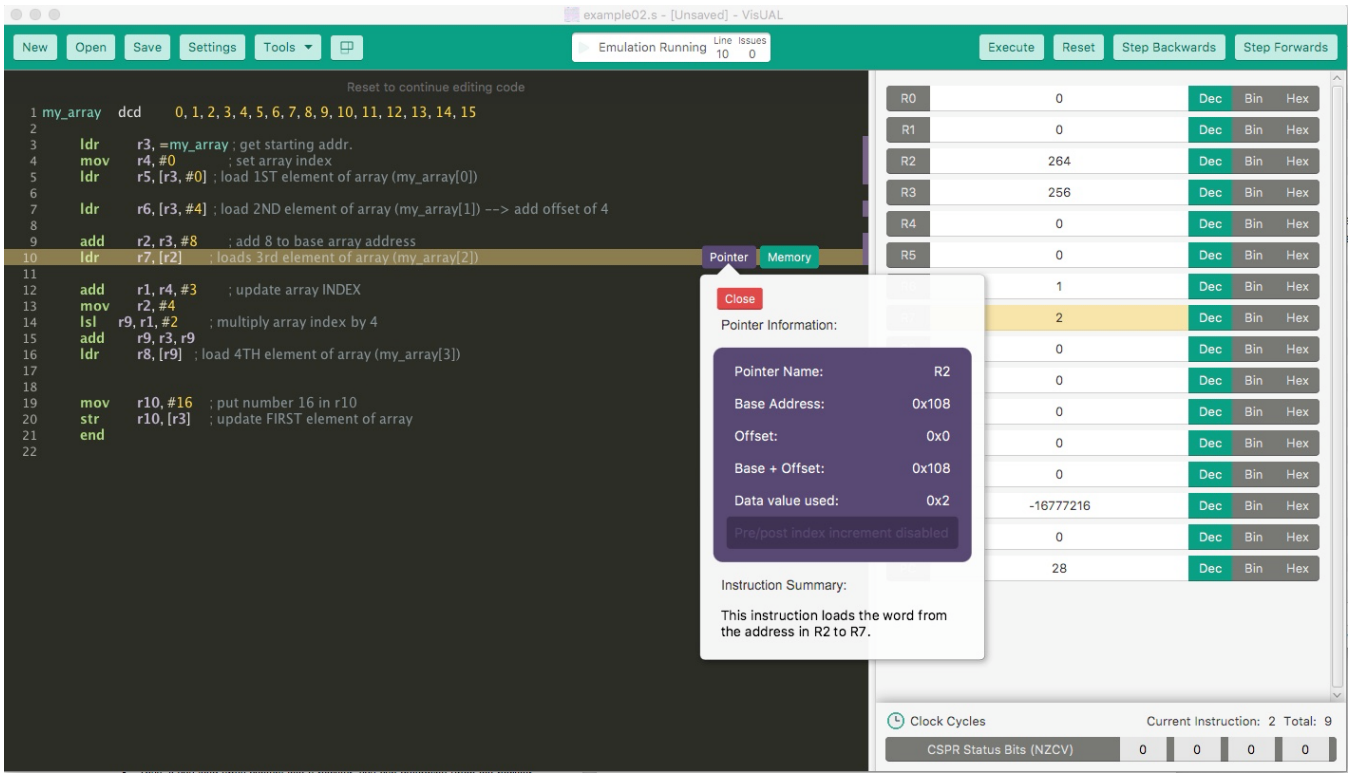

Figure 7: When a load or store instruction is executed, VisUAL provides you with the ability to view what sources – i.e., a base register and offset – are used to calculate the address that data will be loaded from or stored to. (Just click on "Pointer".)

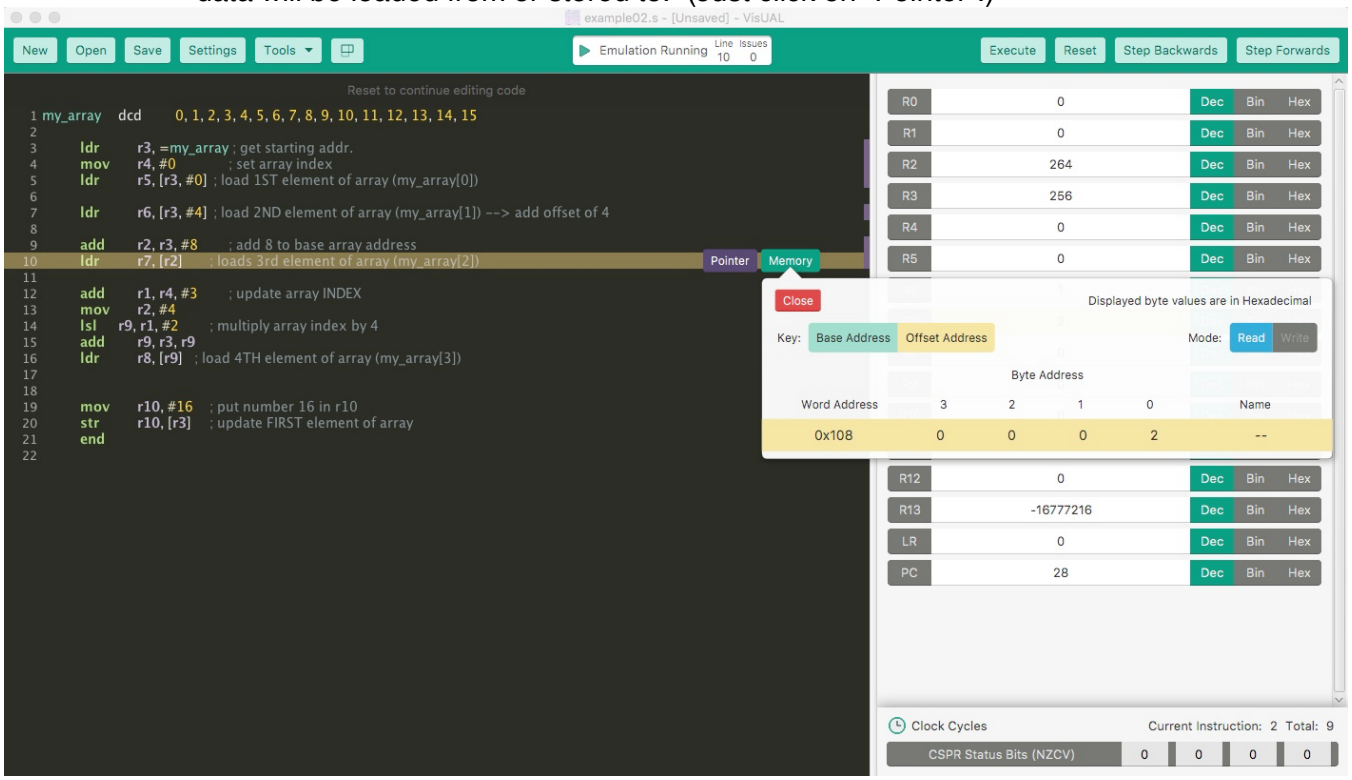

Figure 8: When a load or store instruction is executed, VisUAL will also allow you to easily see the *content* of the address from which data is loaded from or stored to. (Just click on "Memory".)

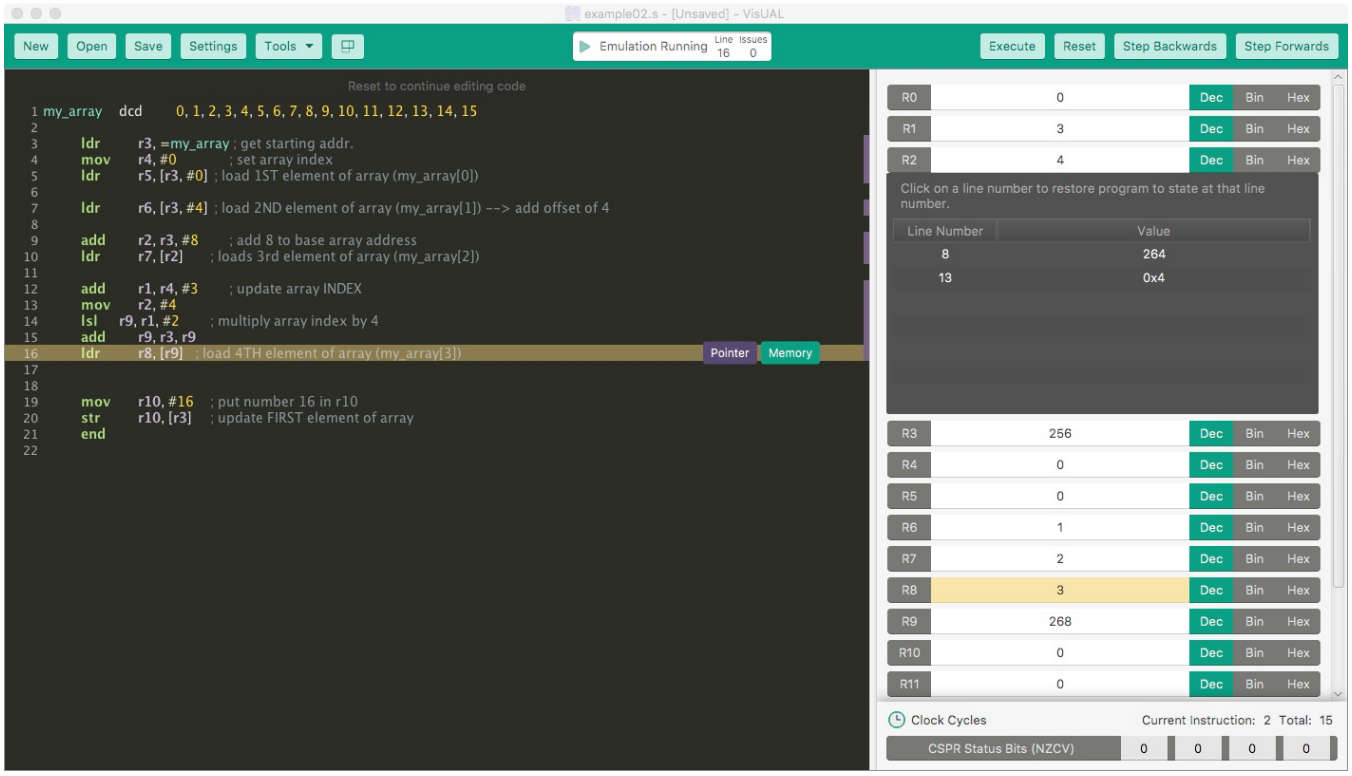

Figure 9: It is also possible to view how the state of an individual register has changed over time. Simply click on the register number, and you can see what instruction changed the value of a register and what it changed it to. You can essentially "rewind" your code by clicking on the line number. (All of this information can be useful for debugging!)

- In the Tools menu, you can pull up another window to selectively view memory content.

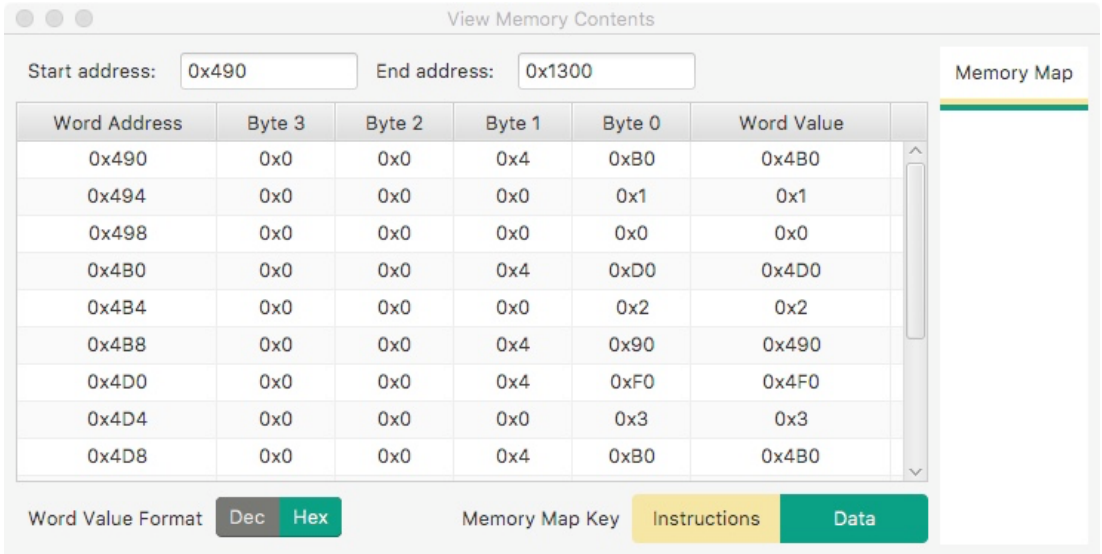

Figure 10: A snapshot of the memory view; you can define a range of addresses to look at. Only addresses that have been written too/referenced will be displayed.

Note that while you can edit your code in VisUAL, I would recommend editing code in a separate text editor and loading into VisUAL. (I've experienced 1-2 crashes while editing in VisUAL.)

3.2 Example Programs

To help you get started with VisUAL, I've placed some example programs that you can load (and immediately run) in the simulator***. It will probably be helpful for you to quickly step through some of these examples, as it will introduce you to a few syntactical statements that you will need to use in the program that you will write (to be run in VisUAL).***

All example programs are in Sakai – see Resources / Labs / 01 / Examples

The program below executes comparisons and conditional branches – labels should NOT be preceded by colons, or you will get a syntax error.

```
            mov   r3, #50          ; initialize r3 to 50
            mov   r4, #100         ; initialize r4 to 100

            mov   r0, #-7          ; initialize r0 to -7
            mov   r1, #-7          ; initialize r1 to -7
            cmp   r0, r1           ; set flags

            beq   next             ; if r0 and r1 equal, goto end

            add   r2, r3, r4       ; this instruction will be skipped

next        cmp   r3, r4           ; compare r3 and r4
            blt   next1            ; if r3 < r4 -- it is -- goto next1

            add  r2, r3, r4        ; this instruction will be skipped

next1       bge     stop           ; if r3 >= r4, goto the end

            add     r2, r3, r4     ; this instruction will NOT be skipped

stop
            end
```

Note that `end` is reserved in VisUAL to facilitate a clean end to the program.

The program below will load values from a pre-defined array in different ways:

```
my_array    dcd   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

            ldr   r3, =my_array    ; get starting address of my_array

            mov   r4, #0           ; set array index
            ldr   r5, [r3, #0]     ; load 1ST element of array (my_array[0])

            ldr   r6, [r3, #4]     ; load 2ND element of array (my_array[1])

            add   r2, r3, #8       ; add 8 to base array address
            ldr   r7, [r2]         ; loads 3RD element of array (my_array[2])

            add   r1, r4, #3       ; update array INDEX to 3
            lsl   r9, r1, #2       ; multiply array index by 4
            add   r9, r3, r9       ; add base address of array to 3*4
            ldr   r8, [r9]         ; load 4TH element of array (my_array[3])

            mov   r10, #16         ; put number 16 in r10
            str   r10, [r3]        ; update FIRST element of array

            end
```

Be sure that you understand: **(1)** how you can load data into memory **(2)** how you can get the starting address of a data array, and **(3)** how to define data in the array. Note that there are many ways to calculate a proper array address as we have discussed in class.

The program below will load some array data and call a function:

```
array       dcd   17, 24, 25, 35
            ldr   r3, =array              ; get starting address of array
            mov   r4, #0                  ; set array index

            ldr   r1, [r3, r4]            ; load first element of array

            add   r4, r4, #4              ; update array index
            ldr   r2, [r3, r4]            ; load second element of array

            bl    proc_call               ; call test procedure

            sub   r5, r4, #20             ; do any operation

            end                           ; exit cleanly

proc_call   adds  r0, r1, r2              ; do any operation
            mov   r15, r14                ; restore program counter
```

Be sure that you understand how to restore the value set by the `bl` instruction after a procedure call is completed.

How to check your code

An easy way to check your code is to simply traverse your list after you have sorted it and load each data word into a register. You can then look at the register history to see if the values are in sorted order. Below I have included screenshots that (a) list the code we used to traverse the list and (b) the history of register R1 (that we used to load each data element). Regarding the latter, note that R1 was used for multiple purposes. However, line 132 is used to load the value in the list for the purposes of checking. Thus, you simply need to look at the values associated with this line – which are in order in this example.

```
124 Print
125         mov     r6, r5
126         ldr     r1, [r6, #4]    ; first value in list
127         ldr     r6, [r6]        ; check if next is null
128         cmp     r6, #0
129         beq     end_print       ; only one in list
130
131 print_loop
132         ldr     r1, [r6, #4]    ; next value
133         ldr     r6, [r6]        ; address of next element's next
134         cmp     r6, #0          ; end of list
135         beq     end_print
136         b     print_loop
137
138 end_print
139         end
```

Figure 11: Code used for copying the sorted, no-duplicates list into R1

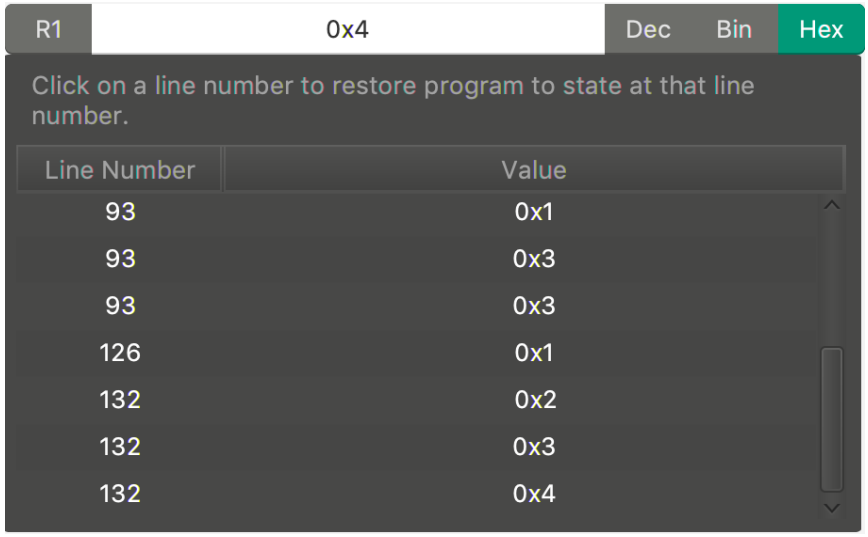

Figure 12: Click on R1 in the register pane on the right of the simulator to view history

What to Turn In

- Put your code for lab01.s in the dropbox of **one** group member. To do this:
  - Login to a student machine – e.g., student06.cse.nd.edu
  - Go to the course dropbox directory; you can type:
    - `cd /escnfs/home/<YOUR NETID>/esc-courses/fa21-cse-30321.01/dropbox`
  - You can then go to your directory – or the directory of the person submitting the lab and make a submission directory
    - `cd <your netID>`
    - `mkdir Lab01`
  - Finally, you can copy your file to the directory
    - E.g., use the `cp` command to copy the file to the directory that you created in `dropbox`
- Be sure that your code runs in VisUAL – as we will use it to test your code with a different array
- Please include a README files that simply says "everything works", or that makes notes of anything that may not work. **Also, please include the names of all group members in the README file as well.**
- No report is required for this lab.

Grading Rubric

If your code runs with (i) no errors, (ii) the list is sorted correctly, and (iii) you "swap pointers not values", you will receive full credit.

If your code is not completely correct, we will grade the lab in "stages"…

- If the list is built correctly, the INSERT function works, etc., this is worth at least 65/100 points
- Next, if the sort works correctly, and the SWAP function is correct, you will earn another 25 points – for a total of 90/100
- Lastly, the delete function is worth 10/10 points

Of course, there are other cases that may come up that are not described here.

- For example, your group's BUILD/INSERT might be correct, and the sort might almost work … this would likely be worth 85/100 – 65/65 points for the BUILD/INSERT and 20/25 for the SORT/SWAP.

In short, even if your code is not completely right, turn it in! Also, document what works and what does not work to the best of your ability so that we can give you as much credit as possible.

Finally, I'd suggest working through this assignment "one piece at a time" – i.e., first get your list built correctly, and then worry about sorting it. The, get the list sorted correctly and then work on deleting duplicate entries… You should also probably start with a small list – i.e., just 3 to 4 times (or even 2 items) to check/debug your code.