Beta Assembly Language

When entering numeric values in the answer fields, you can use integers (1000, 0x3E8, 0b1111101000), floating-point numbers (1000.0), scientific notation (1e3), engineering scale factors (1K), or numeric expressions (3*300 + 100).

Useful links:

- Introduction to BSim
- Summary of Instruction Formats (PDF)
- Beta Documentation (PDF)

Problem 1. Design Problem: Bubble sort

Please see the instructions below.

Use the Bsim instance below to enter your code. To complete this design problem, you should assemble your program, then run the simulation to completion. The built-in test will either report any discrepencies between the expected and actual outputs, or, if your code is correct, it will record the test passed.
To enable the online system to check this answer, first run the design tests provided by the tool.
Open BSim in a new window

Instructions

Suppose you have an array of integers and wish to sort them into ascending order. There are a variety of approaches you might use -- one of the simplest (but slowest) is the bubble sort algorithm. Bubble sort makes multiple passes through the array swapping the previous element with the current element is if the previous element is larger. If no swaps occur on a pass, then bubble sort terminates and the array is sorted in ascending order. If a swap does occur, another pass is made.

Here are the steps in the bubble sort algorithm. `A` is the array to be sorted; actually the value of the symbol `A` is the address of the first element of the array, `A[0]`. The value of the symbol `ALEN` is the number of elements in `A`, called the length of the array. The elements of `A` are `A[0]`, `A[1]`, ..., and `A[ALEN-1]`. The algorithm makes use of several variables: `i` and `swapped`.

1. Set `swapped` to 0.

2. Set `i` to 0.

3. Increment `i`. If `i` is now greater than or equal to `ALEN`, this pass through the array is complete so go to step 5. Otherwise continue with the next step.

4. If `A[i-1]` is less than or equal to `A[i]`, go to step 3. Otherwise swap `A[i-1]` and `A[i]`, set `swapped` to 1, then go to step 3.

5. if `swapped` is 1, go to step 1 and start another pass. Otherwise a pass was completed with no swaps, so the bubble sort is complete.

There are a number of simple optimizations that can be made to this algorithm in order to reduce the total number of steps needed to complete the bubble sort. See the Implementation section of the Wikipedia article on bubble sort for more details.

Your task is to write an implementation of bubble sort in Beta assembly language. You'll enter your code in the BSim window on the lab page. See the BSim documentation on how to assemble and run your code for testing.

Here's the template you'll see in the "Bubble_sort" tab of the BSim window. It includes `beta.uasm`, which describes the Beta instructions to the built-in assembler. The initial `BR(STEP1)` causes the Beta simulator to start executing your code at the instruction labeled "STEP1:". Then the array `A` is defined, occupying 12 words in memory and the value of the symbol `ALEN` is set to the length of `A`.

```
.include "beta.uasm"

       BR(STEP1)   // start execution with Step 1

// the array to be sorted
A:     LONG(10) LONG(56) LONG(27) LONG(69) LONG(73) LONG(99)
       LONG(44) LONG(36) LONG(10) LONG(72) LONG(71) LONG(1)

ALEN = (. - A)/4    // determine number of elements in A

// Please enter your code for each of the steps below...

STEP1:
       ...
STEP2:
       ...
STEP3:
       ...
STEP4:
       ...
STEP5:
       ...

// When step 5 is complete, execution continues with the
// checkoff code.  You must include this code in order to
// receive credit for completing the problem.
.include "checkoff.uasm"
```

Figure out the appropriate Beta instructions to use to complete each of the five steps described above. There are some notes below, which you may find helpful. To test your code, click the Assemble and correct any syntax errors reported by the assembler. If assembly succeeds, you'll be switched from the Editor window to the Simulation window.

You can use the simulation controls to step through your program one instruction at a time (⏯), execute instructions at a steady pace (▶), or execute instructions very rapidly (⏭). You can see the values in each register and memory location, so it's pretty easy to figure out if your code is doing what you expect!

Executing one instruction at a time might be a good way to debug your code for the first pass or so. Once your code completes, execution continues with the checkoff code, which uses the console pane in the simulator to report success or provide an error diagnostic. The checkoff code must complete execution to receive credit for the lab.

Good luck!

Notes

1. It's convenient to use registers hold the value for often-used variables, e.g., `i` and `swapped`. You can make your code easier to read by assigning appropriate names to the registers you've chosen to use to hold various values. Simply add the following assembly language statements to your program:

   ```
   i = R0        // use R0 to hold the value of i
   swapped = R1  // use R1 to hold the value of swapped
   ```

2. To load the 16-bit two's complement constant 1234 into, say, R3, you can use the instruction `ADDC(R31,1234,R3)`. This is a common operation so there's an abbreviation we can use: `CMOVE(1234,R3)`.

   For example, to load 0 into the register with the name `swapped`:

   ```
   CMOVE(0,swapped)   // assumes we used Note 1
   ```

3. Most of the time we want to discard the PC+4 written by the branch instructions into the destination register, so we'll specify R31 as the destination. This is pretty common, so there's abbreviation we can use: instead of writing `BEQ(r2,STEP5,r31)`, we can write `BEQ(r2,STEP5)` and the assembler will supply R31 as the destination register.

4. If you want a branch instruction that always branches, you can write `BEQ(R31,STEP3)` and since R31 always reads as 0, the test succeeds and the branch is taken. There's an abbreviation we can use: `BR(STEP3)`, which assembler expands into `BEQ(R31,STEP3)`.

5. To load the i^{th} element of array `A` into a register, first compute the byte address of the array element and then use the `LD` instruction to fetch the desired value. The byte address of the first (i.e., zeroth) element of array `A` is the value of the symbol `A`. Each 32-bit value in the array occupies a 32-bit word, or 4 bytes. So we multiply `i` by 4 to convert from the array index to the appropriate byte offset from the beginning of the array. We can then use the address arithmetic built into the `LD` instruction to combine the value of the symbol `A` with the byte offset of the element we want:

   ```
   MULC(R0,4,R2) // i in R0, convert index into byte offset
   // load address is Reg[Ra] + sxt(16-bit) literal
   LD(R2,A,R3)   // loads A[i]
   LD(R2,A-4,R4) // loads A[i-1]
   ```