How Computers Work Problem Set 1

Problems 1-3

Among the many algorithms that can be used to sort items, the **merge sort** is one of the easiest to understand. Let's say you have a pile of items you want to sort. Since this is hard, break the pile roughly in half into two piles. Sort each of those piles to put it in order. Then, merge the results into a sorted pile by repeatedly picking off the appropriate object from each of the two previously sorted piles. Of course, this is a recursive definition, so it is important to know when a pile is so small it does not need sorting.

Consider the following code for a merge-sort implementation in :.

```
void copy(int *from, int *to, int n)
{
  while (n--)
    *to++ = *from++;
}

void merge(int *array_a, int n_a, int *array_b, int n_b, int *array_c)
{
  while (n_a && n_b)
  {
    if (*array_a > *array_b)
    {
      *array_c++ = *array_b++;
      n_b--;
    }
    else
    {
      *array_c++ = *array_a++;
      n_a--;
    }
  }

  while (n_a)
  {
    *array_c++ = *array_a++;
    n_a--;
  }

  while (n_b)
  {
    *array_c++ = *array_b++;
    n_b--;
  }
}

void sort(int *array, int n)
{
  int *array_a, *array_b, *array_c;
  int n_a, n_b;

  if (n > 1)
  {
    array_a = array;
    n_a = n / 2;

    array_b = array + n_a;
    n_b = n - n_a;
```

```
        sort (array_a, n_a);
        sort (array_b, n_b);

        array_c = allocate(n);

         merge (array_a, n_a, array_b, n_b, array_c);

        copy (array_c, array, n);
      }
    }
```

Problem 1 - To be done alone

On a separate sheet of paper, copy the program above with a comment (enclosed in /* and */ , as is typical in C) to the right of each line of the program clearly explaining its purpose. Be very specific, as if you are teaching someone else how the program works. Use as much space per comment as necessary. Don't simply write down **what** the line does, but rather **why** it is there.

Problem 2 - To be done with colleagues

Using the Beta Instruction Set and software calling conventions, hand-compile the copy() routine into Beta instructions.

Problem 3 - To be done alone

Consider the allocate(n) routine used in the merge subroutine.

A) Explain why the allocate(n) routine is necessary.

B) The allocate routine could be simply translated into a few Beta instructions that include a call to the allocate macro-instruction for allocating stack space. Write out this translation and explain why allocating stack space, instead of more permanent storage like heap space, is sufficient.

C) Notice that there is no de-allocate routine. Explain how the software calling conventions for the Beta make this unnecessary.

Problem 4 - Lab Assignment - Hacking fractals with BETASIM

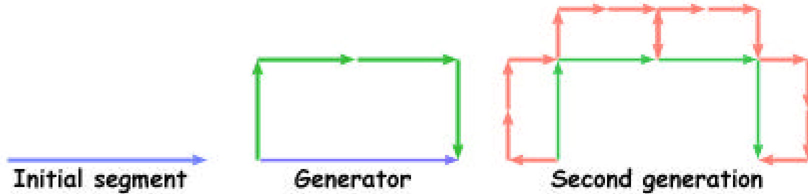
The Betasim code uses Java 1.1, which means that Internet Explorer version 4.0 (PC and Mac) will run it, as will Sun's JDK1.1 appletviewer program. Netscape, except on PCs, will NOT run the code unless you can find a Java 1.1 updater (I can't). Specifically, the code is known to run correctly on the computers in the lab, and support is NOT OFFERED for any other computers. If it runs, great. If not, come to the lab. You should plan to come to the lab anyway, to get help from LAs, and talk about your ideas with other students.

A note on working with Betasim: Save early, and save often! If the program crashes and you lose your work, we can't recover it. To save code that you type in the code window, use the File>Save menu command in the code window. To save your Beta schematic and ROM code, use the File>Save menu command in the schematic window. Saving your code does NOT save the

schematic; saving the schematic does NOT save any changes you've made to the code.

Background on Fractals

You've all done this already in SICP, but just to refresh your memory...



A fractal is a self-similar curve. You construct one by starting with a simple line segment. Replace the line segement with the "generating curve", scaled so that the ends of the generating curve match the ends of the line segment. Repeat for each of the line segments you just added. If you keep repeating, the line segments get smaller and smaller, and you've got your fractal!

C code

```
void fractal(int x1, int y1, int x2, int y2, int level) {
    if (level==0) {
        drawLine(x1, y1, x2, y2);
    } else {
        int xdist= (x2-x1)/2;
        int ydist= (y2-y1)/2;
        level= level-1;
        fractal(x1, y1, x1+ydist, y1-xdist, level);
        fractal(x1+ydist, y1-xdist, x1+ydist+xdist, y1-xdist+ydist, level);
        fractal(x1+ydist+xdist, y1-xdist+ydist, x2+ydist, y2-xdist, level);
        fractal(x2+ydist, y2-xdist, x2, y2, level);
    }
}
```

We've given you the initializing and drawLine assembly code already (phew!). All you have to do is write the assembly code for the fractal function.

Things to think about:

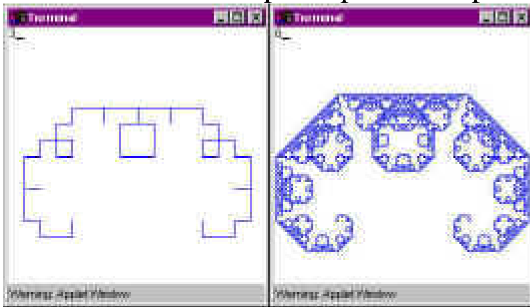
- You should write the entire function in assembly on paper before you try entering it in Betasim. It'll take less time to type in and debug.
- This is obviously a recursive function. You'll probably want to follow the calling conventions for the stack that we talked about in class. If you want to make the code more efficient, read on. Otherwise, just make sure to save everything you use on the stack.
- Is it better to make the callee responsible for saving and restoring all of the registers, or should the caller be responsible for saving the value before makeing the call?
- Which registers do you really need to save?
- How efficient can you be?

How to enter and test your code:

- Before you enter your code, you should try **writing the code on paper**. Use a printout of the framework code to help you get started. A list of beta macros should also be of some use. We

also have a tutorial, if you're not sure how to use the Betasim program.

- Try running your code with one or two of the recursive fractal calls before you add the code for all four recursive calls. This will get your code working first with less editing and typing. Then you can add the other parts using the same pattern.
- The code editor does not yet support cut-and-paste. Sorry.
- **To test your code**, click reset, click run, and note that the code will go into a tight loop waiting for input. Go to the terminal window and type a single digit, say 2. This is the recursion depth you request. To try a different depth, click reset a few times and enter a new recursion depth.
- You might try selecting the **go fast** option from the Control Panel.
- The code will replace one line with four lines for each generation. The original line, from the previous generation, will not remain (in the above figure, this is shown in different colors).
- Here are some example expected outputs (for levels 3 and 6, respectively):



For the Real Geeks

How the terminal works:

The input and output channels to the terminal are "memory-mapped" to some high address words in memory. In particular, INPUTDATA and OUTPUTDATA are at addresses -32 and -24, respectively. Memory-mapping is a technique where special input/output channels look like memory at specific addresses to the code. In reality, there is special hardware that recognizes the memory-mapped addresses, and routes the data to and from the particular input and output channels.

If you write to OUTPUTDATA, the ascii character corresponding to the data value appears on the terminal. If you type at the terminal window, the characters you type are queued up and can be read sequentially from the INPUTDATA address. If tehre is nothing more to read in the queue, then reading from INPUTDATA returns 0. Trying to write to INPUTDATA or trying to read from OUTPUTDATA doesn't do anything.

For the *really* adventurous, you can play with interrupts. To enable the input interrupt, write INTERRUPT_ENABLE (0x8000) to address INPUTCMD (-31). Now, when someone types a character at the terminal, the computer traps to address 0x80000003. To clear the interrupt, write a 0 to INPUTCMD, and then write INTERRUPT_ENABLE to INPUTCMD to re-enable interrupts. There is also a timer you can play with. Write the number of instructions you'd like to execute between timer interrupts into memory address TIMERDATA (-16), and then write INTERRUPT_ENABLE to TIMERCMD (-15). The trap address for the timer is 0x80000002. Note that interrupts will be ignored as long as the supervisor bit is set, so you must be in user mode in order to get interrupted. Also note that you can't write to or read from the negative gizmo addresses *unless* the supervisor bit is set. Have fun!

Getting checked off

Come into lab and show your code and results to a staff member. You will also have to show the other parts of the lab to the staff, so you may do so all at once, or as you complete each item, whatever is most convenient for you.