Dates

Early on Thu 14-April-2022

The long lead time on this doesn't mean you can safely delay starting the lab.

Due on Fri 15-April-2022

There is no late for lab 6! The graders are graduating seniors and you need to be studying for the final exam which is worth twenty times more than this lab.

Topics

- Using the full range of intel addressing modes
- Register allocation
- More size-aware coding (it's not all quads)
- structs and arrays
- loops

Contents

Overview – Automatic Speech Recognition

This lab will dabble in techniques used in speech recognition. It will borrow heavily from speech recognition, but may vary from state of the art in order to make a better lab; one that students can do and that teaches different aspects of assembler. In automatic speech recognition (ASR), the incoming analog signal is digitized using a certain number of bits per sample at a particular sampling rate. Historically, telephone systems allowed an 8 kHz sampling rate at 8 bits of data per sample. Music CDs are higher fidelity; they sample at 44.1 kHz with 16 bits of data per sample. The higher sampling rate allows the capture of higher frequencies. Using more bits per sample gives greater signal to noise ratio.

When performing ASR, samples are often grouped into frames. A typical number would be 10 ms worth of samples in a frame. The frames usually overlap, but that won't be a concern for us. Our concern is two of the many possible computations performed on a per-frame basis. These computations are called "features" in ASR. Feature selection is critical to good system performance. Our features are the number of zero-crossing and the amount of energy in the frame.

A high number of zero-crossings typically indicates a preponderance of noise. Consonants tend to exhibit low energy levels leaving vowels to provide most of the energy of an utterance.

Our task

Our lab will analyze an array of frames of sample data. Each frame will be held in a C structure. Our lab will compute the number of zero-crossings and the energy of each frame and update the frame structure with those numbers. It will look for the frame with the highest number of zero crossings. We will need a few functions to do all of this.

You will need to write two functions, search and analyze. They will deal with Frame structures, given below.

Frame structure

Here is the declaration for a frame structure:

```
typedef struct {
        long frame_id;
        short zc;
        long energy;
        short samples[64];
} Frame;
```

You will need to determine the size of a frame structure. You will need to determine the offsets for the various members. Search will be handed an array of frame structures. Analyze will be handed a pointer to a single frame structure.

Function Signatures

The lab has various functions that we deal with:

```
/* to be written as part of the lab */
short analyze(Frame *f);
Frame *search(Frame *array, int count);

/* a function provided to you */
short evaluate(Frame *f);
```

You will write analyze and search. The evaluate function is given in the zip.

Functions

Analyze

Basics

The analyze function will be given a pointer to a frame structure. It will compute the number of zero-crossing and the energy for the frame and update the structure with those numbers. It will return the zero-crossing count. Here is the signature:

```
short analyze( Frame *f);
```

We will count zero crossings whenever the sign of a sample is different than the sign of the previous sample. To make life easier, we will deem zero as positive.

To compute energy, we will square the amplitude of the sample and sum up those squares.

Size considerations

The number of zero-crossings can't exceed the number of samples. The fastest zero-crossing rate is when successive samples alternate sign. For telephony-based ASR, a frame might be 80 samples. 10ms at 44.1 kHz (trying to recognize speech off a CD quality input) is 441 samples. We can't fit 441 into a byte but it easily fits into a C short or int. Our "frames" will have only 64 samples, but that is only to make it easier to test.

Squaring the amplitude needs serious consideration. A sample is 16 bits in size, so the sample squared needs 32 bits. We are going to sum a bunch of those squares, and each time we add we might need one more bit. This might make a 64-bit C long seem too small; add up 32 or so maximum amplitude squared values and we will run out of bits long before we get to 441 of them or even 64 or them. Please note that real signals do not sustain maximum amplitude for appreciably long periods of time; they are composed of sinusoids and they reliably go from positive peak values down through small values on their way to their negative peak value. We will store the total energy in a C long. The data we have will not overflow a C long, and you do not have to check for overflow. You do have to be very careful about what sizes are used in the squaring process; the easiest way to get good results is up-size the sample from a C short to a C long and then do all math in C longs.

Constraints

Analyze does not call any other functions, so register choices should be obvious.

Analyze will need to read and write the frame structure. Your code is restricted to a **single** memory touch on any address.

Testing

You can test your analyze code with the attest executable. There is a build target for it in the supplied makefile.

```
[kirby.249@cse-sl2 lab6]$ make -r atest
make: `atest' is up to date.
```

```
[kirby.249@cse-sl2 lab6]$ atest
```

Search

Basics

Search is given an array of frames and a count. It gets each frame analyzed and checks the returned values to keep track of the one with the highest number of zero-crossings. When done searching, it will print some results. It returns the address of the frame with the highest count.

```
Frame *search(Frame *array, int count);
```

Search will loop through the array (either forward or backwards) and call evaluate with the address of each frame structure. Note that evaluate has the same signature as analyze; evaluate does the job of getting analyze called and it returns the same return value that analyze provided. Evaluate takes care of printing out the updated structure so that you don't have to.

Search needs to track the address of the frame with the highest count as well as the highest count. If the current frame has a higher count, it becomes the new frame with the highest count and its count becomes the new highest count.

Search will need to be able to figure out the address of the next structure in the array.

Constraints

In the search loop, search does not need to go to memory at all.

At the end, search will need to go to memory to feed the final print call that has the frame number and the number of zero-crossings.

The search function has a function call inside a loop – it clearly needs callee-saved registers. There are 5 available callee-saved registers (rbx, r12-r15). This is enough. Don't forget that you can use caller-saved registers for temporary values that don't have to survive a function call.

Testing

Do search after you get analyze working correctly. Use the lab6 executable in the makefile to test it.

Output

Here is the lab output for a loop that goes in increasing subscript order:

```
Evaluating frame 8002003001
analyze returned 36
Frame 8002003001 has 1592896 energy and 36 zero-crossings.

Evaluating frame 20123123123
```

```
analyze returned 63
Frame 20123123123 has 64 energy and 63 zero-crossings.

Evaluating frame 7000123456
analyze returned 7
Frame 7000123456 has 345008 energy and 7 zero-crossings.

Search selected frame 20123123123 with 63 zero-crossings

Lab 6: frame 20123123123 is best
```

Here is the output for searching in reverse order:

```
Evaluating frame 7000123456
analyze returned 7
Frame 7000123456 has 345008 energy and 7 zero-crossings.

Evaluating frame 20123123123
analyze returned 63
Frame 20123123123 has 64 energy and 63 zero-crossings.

Evaluating frame 8002003001
analyze returned 36
Frame 8002003001 has 1592896 energy and 36 zero-crossings.

Search selected frame 20123123123 with 63 zero-crossings

Lab 6: frame 20123123123 is best
```

Here is the output for atest

```
Atest is calling evaluate.

Evaluating frame 7000123456
analyze returned 7
Frame 7000123456 has 345008 energy and 7 zero-crossings.

Done.
```

Bonus

Don't even think about this until you get the lab working.

Branchless code

In the loop in analyze, it is possible to write branchless code. The loop will have to branch, but you can get 1 bonus point if you can count zero-crossing without branching

Easing register pressure

With 5 available calle-saved registers available to use in search, you have enough. Can you write search in 4 or fewer callee saved registers? If you resort to touching memory when a callee-saved register is available, not only do you *not* get the bonus, but you lose 3 points for not using registers properly. But if you can tighten your code to use just 4, you get a bonus point.

Shims

You are required to use print and not printf in your code. They have identical signatures. There are 3 shims in the zip, but you only have to deal with print.

- print is used to call printf. It calls printf for you.
- a_shim is used to validate the analyze behaved well.
- s_shim is used to validate the search behaved well.

The supplied lab6 code call s_shim and s_shim calls search.

Your search code will call evaluate, evaluate will call a_shim, and a_shim will call analyze.

Pay attention to any messages generated by the shims, they can save you big points.

Register Allocation

Register allocation policy is graded – similar to lab 5.

Required Stuff (same as lab 5)

All of your functions **must** create a stack frame upon entry and tear it down upon exit.

The first usage of any register in a function **must** have a comment telling what that register means. For example:

```
xorq %rax, %rax # rax is both a subscript and return value at the same time.
```

This is usually near the top of your function. It is your pocket guide to the registers. In your comments you should refer to the registers not by name but by what they mean.

Comments (same as lab 5)

Comment your code. Comments should say things that are not obvious from the code. *In assembler you could easily have something to say for every line of code. You could easily have more comment lines than code lines.* Comment what each register holds. Comment about how the operation has a higher level **meaning.** Try to avid comments that say the exact same thing as the code.

Put your name and the assignment number in your initial comments. Also add the statement that says that you wrote all of the code in the file (see below). Or you can forget it and get a **zero** on the lab.

Readme

As always, create a text README file, and submit it with your code. All labs require a readme file. Include:

- Your name
- Hours worked on the lab
- Short description of any concerns, interesting problems, or discoveries encountered. General comments about the lab are welcome.

Submission

No surprises here. Your zip file needs:

- A readme file
- All of your .s files
- Makefile
- The supplied .c files, .h file, and shim .o files

Be sure to add this text to ALL of your .s files:

```
# BY SUBMITTING THIS FILE AS PART OF MY LAB ASSIGNMENT, I CERTIFY THAT
# ALL OF THE CODE FOUND WITHIN THIS FILE WAS CREATED BY ME WITH NO
# ASSISTANCE FROM ANY PERSON OTHER THAN THE INSTRUCTOR OF THIS COURSE
# OR ONE OF OUR UNDERGRADUATE GRADERS.  I WROTE THIS CODE BY HAND,
# IT IS NOT MACHINE GENRATED OR TAKEN FROM MACHINE GENERATED CODE.
```

If you omit a required file, you get **zero** points.
If you fail to add the above comment you get **zero** points
If the make command as given generates any warnings or errors you get **zero** points

Every file you hand edit needs your name in it.

This is one of the easiest labs to get full marks on, don't blow it.