Contents

Version 2

The size directive is required. The “pocket guide to registers” is clarified – see the comments section.

Dates

There are no prototypes for assembler labs. It is a very good idea to build a little, test a little as you go.

Early: Saturday, 2-November-2024 at 11:58 PM
On Time: Tuesday, 5-November-2024 at 11:58 PM
Late Cutoff: Wednesday, 6-November-2024 at 11:58 PM

Lab 5: Assembler in Three Functions

Lab 5 introduces:

- Assembler programming
- Calling functions
- Loops
- Decisions
- Arrays
- Pointers
- Register allocation
- Stack details

The short form of what goes in in lab 5 is: Convert an array of what might be 16-bit linear PCM data into integer db values. (16-bit linear PCM is how "Red Book" CD audio data is encoded.) Along the way, track the largest and smallest values. Do this using three functions that you write and some supporting functions that will be given to you.

The Three Functions and Four Shims

Here are the signatures of the three functions you will write. You will write them in assembler. Your code will follow all conventions we have gone over. **Do not write any C code for this lab.** No code that you write will directly call these two functions.

```
int analyze(short sample);

int middle(short sample, int *min_ptr, int *max_ptr);

void convert( short pcm[], int db[], int count);
```

These functions will not be called directly. Instead "shims" will fit between the caller and the function called. They have the same signatures as your code. The shims do various forms of error checking on your behalf.

```
int a_shim(short sample);

int m_shim(short sample, int *min_ptr, int *max_ptr);

void c_shim(short pcm[], int db[], int count);
```

Note the mixture of 32 bit ints, 16 bit shorts, and 64 bit pointers. Your assembler code will need to be aware of sizes.

There is also a shim that goes between your code and printf. That shim is called print. Don't sweat the signature, you are calling it from assembler code. Do **not** call printf directly from your assembler code. Use print just like you use printf.

All of the shims will detect if they were called with the stack meeting the 16-byte boundary rule. They will print an error if the stack was not correctly aligned. Your code **must** follow the 16-byte alignment rule.

These functions are tested by different make targets. The attest target uses analyze and the lab5 target uses analyze, middle, and convert. Do them in that order.

Analyze

```
int analyze(short sample);
```

Use the atest target in the makefile to test your code. You don't need to write middle to write analyze.

At its core, analyze finds the number of significant bits in a 16-bit short. In audio encoding, you get about 6 db of signal to noise ration per bit used in the sample. So we can expect at most 96 db out of 16-bit audio. So we will take the number of significant bits and multiply that by 6 to get a rough approximation of the db level of any given sample.

For positive numbers, the number of significant bits is simply to location of the left most 1 bit in the sample. If the sample has a value of 2 or 3, the result is 2. The values 4-7 give a result of 3.

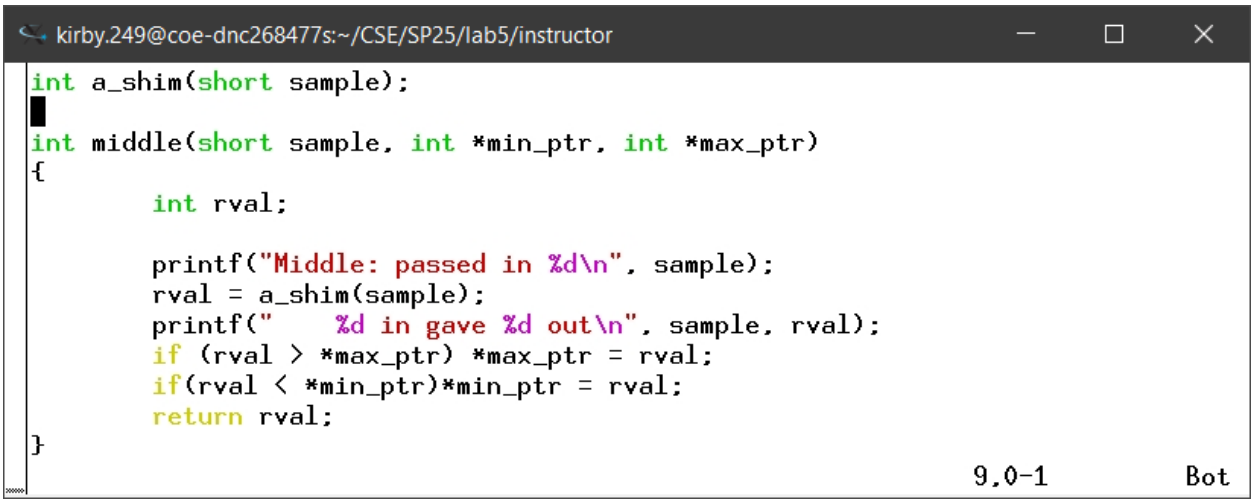
For negative numbers, the result we want is the same as the number for the same magnitude positive number. Numbers between -4 and -7 should give 3. The hard part is the most negative value, which when negated gives back the same value. (The most negative value should give 16 as the number of significant bits for a return value of 96.)

One way is to use a loop to seek the number of significant bits. For a positive number, you could count the number of right shifts it takes to make that number zero.

There are many ways to do the final multiply; imul is the most straightforward, but you could add twice the value to four times the value to get six times the value, with shift providing the x2 and x4. You could also employ lea.

Middle

The function middle exists mostly to teach you about callee saved registers, stack allocation, and just what hog printf happens to be. Its main purpose in life is to call a_shim to return the value it gets from analyze. It also features basic decision making and accessing memory via pointer. Here is what middle should do, expressed in C code:



Do not transcribe this C code as C code!

The presence of a middle.c file is an indicator of academic misconduct. All mercy rule zeroes will be reinstated without discussion and you can expect to wind up in front of CoAM. This is also true of analyze and convert; an analyze.c or convert.c file will be treated as academic misconduct.

This code gives a clear an unambiguous set of requirements for what the function middle is expected to do. Decompose the C code and the implications of every single line and use it to guide you as you write your middle.s file. *Doing this is practice for something you will be expected to do on the final exam.*

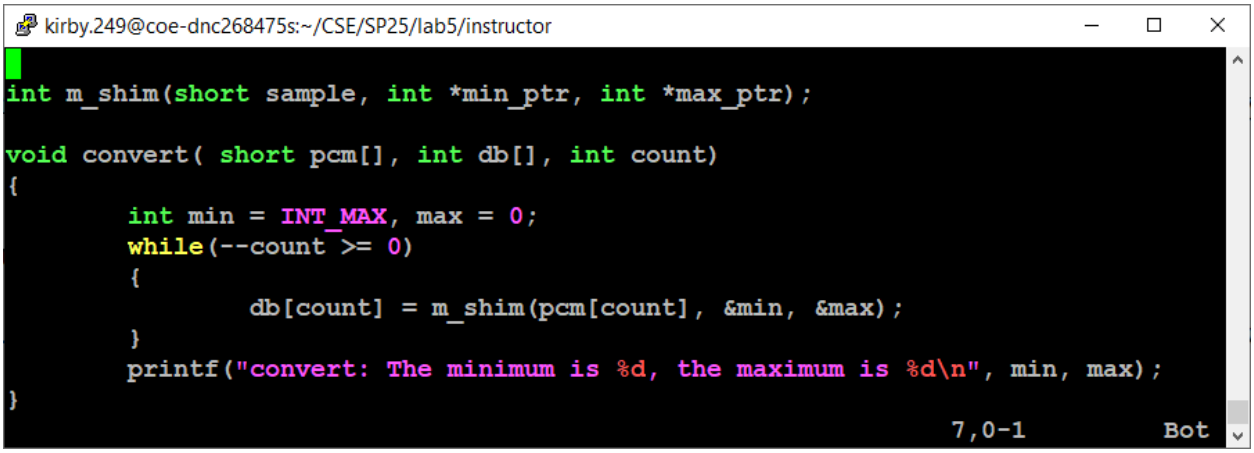
Middle is where you find out if your code follows the 16-byte alignment rule. If need be, you can grow the stack using subtract. Be sure that you understand where to do this and by how much.

Convert

Recall the signature for convert is:

```
void convert( short pcm[], int db[], int count);
```

So convert gets passed in two arrays and the size of those arrays. It will loop through the pcm values to indirectly call analyze to get back db values that it fills into the db array. See the C code below.



Do not transcribe this C code as C code!

Note that min and max are local variables. The code takes the address of min and max and passes that to middle via m_shim. Registers do not have an address, so you can't store min and max in registers. You can use any sufficiently large value in place of INT_MIN. 100 is sufficiently large since the largest value out of analyze is 96.

About this time you might review register allocation and the rules and guidance on what goes where and why. Along with that a refresher on how we lay out the stack might be in order.

Use the lab5 target in the makefile to test your convert code. You will need a working analyze and middle.

Output

For atest:

```
[kirby.249@coe-dnc268475s lab5]$ ./atest
atest: passed in 16384
    16384 in gave 90 out
atest: passed in -16384
    -16384 in gave 90 out
atest: passed in 2
    2 in gave 12 out
atest: passed in -2
    -2 in gave 12 out
```

```
atest: passed in 0
    0 in gave 0 out
atest: passed in 32767
    32767 in gave 90 out
atest: passed in -32767
    -32767 in gave 90 out
atest: passed in -32768
    -32768 in gave 96 out
atest: passed in 1024
    1024 in gave 66 out
atest: passed in -1024
    -1024 in gave 66 out
atest: passed in 1
    1 in gave 6 out
atest: passed in -1
    -1 in gave 6 out
[kirby.249@coe-dnc268475s lab5]$
```

For mtest

```
[kirby.249@coe-dnc268475s lab5]$ ./mtest
Mtest: min =2147483647, max = 0
Middle: passed in 16
    16 in gave 30 out
Mtest: m_shim returned 30
Mtest: min =30, max = 30
Middle: passed in 0
    0 in gave 0 out
Mtest: m_shim returned 0
Mtest: min =0, max = 30
Middle: passed in -32
    -32 in gave 36 out
Mtest: m_shim returned 36
Mtest: min =0, max = 36
[kirby.249@coe-dnc268475s lab5]$
```

For lab5:

The line tagged with green is convert's contribution to the output.

```
[kirby.249@coe-dnc268475s lab5]$ ./lab5
Middle: passed in -1
    -1 in gave 6 out
Middle: passed in 1
    1 in gave 6 out
Middle: passed in -1024
    -1024 in gave 66 out
Middle: passed in 1024
    1024 in gave 66 out
Middle: passed in -32768
    -32768 in gave 96 out
Middle: passed in -32767
```

```
    -32767 in gave 90 out
Middle: passed in 32767
    32767 in gave 90 out
Middle: passed in 0
    0 in gave 0 out
Middle: passed in -2
    -2 in gave 12 out
Middle: passed in 2
    2 in gave 12 out
Middle: passed in -16384
    -16384 in gave 90 out
Middle: passed in 16384
    16384 in gave 90 out
convert: The minimum is 0, the maximum is 96

   PCM    db
 16384    90
-16384    90
     2    12
    -2    12
     0     0
 32767    90
-32767    90
-32768    96
  1024    66
 -1024    66
     1     6
    -1     6
[kirby.249@coe-dnc268475s lab5]$
```

GDB

The first time atest builds, run it under gdb and step through your code. Invoke tui reg general and predict how each register will change before you take each step. Watch the values appear (in hex) in the various registers. Go back to lab 1 and look at all of the gdb commands you were required to use there and refresh them in your memory.

Probable bugs: Reversing the sense of a condition runs very high on the list!

Probable bugs: Mixing up two register names. This language is hot revenge for all those times you might have slacked off and picked a less-than-stellar variable name when you had 32 characters at your disposal.

DO NOT CALL PRINTF TO DEBUG! Trying to debug with printf is negative progress; it makes things worse.

Imagine a grainy blue hologram: "Help me, GDB. You're my only hope."

https://www.youtube.com/watch?v=5cc_h5Ghuj4

We might need more than the ability to see what is in the registers, we might need to be able to examine memory.

https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_55.html

Run gdb on mtest. Set a breakpoint. Then issue the `run` command. Issue the `tui reg general` command. Let's look at memory. To do that, we need an address. The memory we want to examine is in the stack. RSP and RBP point into the stack but in the case of middle, so do rsi (min pointer) and rdx (max pointer). Let's look at the int that rdx currently points to. Use this gdb command:

```
x/1dw $rdx
```

It e**x**amines **1** **d**ecimal format "**w**ord" (which here means 32 bit) starting at the address in **rdx**. It will print the address where it started, a colon, and then the value. Use g for the size "giant" and x for the format to look at an address in hex). Poking at memory this way lets us see the values that get written at the end of the middle function.

Bonus

Short Code Bonus

If the loop that counts bits in analyze is 3 statements, one bonus point. Two points if you can get the bits counted in 2 statements. (This doesn't include the multiply at the end or preprocessing the sample before your code gets down to counting bits.)

Main Bonus

One bonus point for implementing the main function and the pcm array given in lab5.c in assembler. Compare bonus.c to lab5.c to see what was commented out. This takes some serious doing. You do not have to implement the function output in assembler to get this point. Note that the array is static storage class. You will need to create main.s for your main function and use the bonus target in the supplied makefile.

Register Allocation

Register allocation policy is graded.

One of the required functions is leaf-level and the other is not. This guides your register allocation policy.

You really do want a firm grasp on register allocation before you take the final exam, so work it out here and in lab 6. Before you write any code, figure out how many "variables" you will need and assign them to registers of the appropriate type.

Required Stuff

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

All of your functions **must** have a .size directive.

All of your functions **must** have a "pocket guide to the registers" in comments, see below.

Comments

Comment your code. Comments should say things that are not obvious from the code. *In assembler you almost always 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 – this might get you points off. In your comments you should refer to the registers not by name but by what they mean.

Near the top of every function, write out in comments what every register the function uses will mean, other than rsp and rbp. This is your pocket guide to the registers when writing the function.

Here is an example that has no bearing on this lab of what needs to be at the top:

```
# rax holds the max count of successive 1 bits that we will return
# rcx holds the current count of successive 1 bits in the current
#     burst of one bits.
# rdi holds the value we will get the count of the longest unbroken
#     string of one bits from.
# rdx holds the "travelling" bit that moves from right to left,
#     probing the input value for 1 bits.
# rsi will hold either a 0 or 1 to indicate that the traveling bit
#     found a 1 bit in the value.
```

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, modified to add a zip target and your name
- The supplied files (such as shims) needed to make lab5, mtest, and atest build (plus any others for the second bonus)

Be sure that the zip target in the makefile self-tests all executables from the contents of the zip file. Any zip file that fails to build gets a zero. Any makefile that doesn't self-test the zip is marked **late** regardless of when it was turned in.

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.