Lab 6 Structs, Function Pointers, Multiply & Divide

Contents

Lab 6 Structs, Function Pointers, Multiply & Divide 1
Version 2 2
Comments & Labels expanded to include labels. 2
Dates 2
Early by 11:58 PM, Wednesday 20-Nov -2024 2
On Time by 11:58 PM, Friday 22-Nov-2024 2
There is no late Lab 6 – study for finals! 2
Introduction 2
Register Pressure 2
The Structs 2
The analyze function 3
Computing and Summing Energy 4
Average Energy and Using Divide 4
Valid Frame 4
Memory Touches 4
Plan of attack 5
The search function 5
Search's Task List: 5
Plan of Attack for Search 6
Shims 6
Output 6
atest 6
lab6 8
Bonus 9
Register Allocation 9
Required Stuff 9
Comments & Labels 9
Readme 10
Submission 11
Machine Generated Code 11

Version 2

Comments & Labels expanded to include labels.

Dates

Early by 11:58 PM, Wednesday 20-Nov -2024

On Time by 11:58 PM, Friday 22-Nov-2024

There is no late Lab 6 – study for finals!

Introduction

This lab involves:

- Structs
- Arrays & Pointers
- Divide
- Multiply
- Careful use of registers

You will write two files of assembler code, analyze.s and search.s for this lab. The supplied zipfile has everything else you need, but the makefile will have to be customized with your zip target and your name.

The idea behind the lab is to do some simple processing on audio data. We will have an array of 16-bit linear PCM data. Our code will compute the energy values of the first few samples of the signal in order to do some levels checking. From the energy values, we will compute the average energy and total energy of those initial samples. If the average energy falls within a range given, the data is valid for use. Our code will search for the data with the highest average energy and it will also track the minimum average energy from all of the valid data. Our search will ignore invalid data.

Register Pressure

Both functions are under register pressure. It is easy to run out of caller saved registers in analyze. It is quite hard to stay in 5 callee saved registers in search. One way to ease this is to run loops towards zero instead of from zero and counting up. The flags will tell you when a register hits zero and they will tell you when it goes negative. The reference code runs the loops backwards – we saw this in lab 5 as well – in order to save one register. Performance is often a game of small gains adding up to big gains. Changing how you code C to run loops towards zero can buy you a small gain.

The Structs

The following code is in frame.h:

```
#define FRAME_ARRAY_SIZE (16)
struct Frame
```

```
{
        int count;
        bool valid;
        short *pcm;
        long total_energy;
        int avg_energy;
        int energy_array[FRAME_ARRAY_SIZE];
};

struct Dataset
{
        int min;
        int max;
        struct Frame f;
};
```

Note that there are two very different arrays in the frame structure. In one case we have a pointer to data that lives elsewhere. In the other case, we have data that is allocated inside the struct. The way we get to them will be different (and was commonly missed on a previous final exam). *They look the same in C but not in assembler!*

```
sample = fp->pcm[i];
fp->energy_array[i] = e;
```

You need to know what the offsets (displacements) are for the members of the structs. For the final exam, be able to compute these for yourself, but here they are:

```
Frame:
 The offset for count is 0
 The offset for valid is 4
 The offset for pcm is 8
 The offset for total_energy is 16
 The offset for avg_energy is 24
 The offset for energy_array is 28

Dataset:
 The offset for min is 0
 The offset for max is 4
 The offset for f is 8
```

The analyze function

```
bool analyze(struct Frame *fp, int lower_threshold, int upper_threshold);
```

This is a different analyze than we saw in lab 5! Analyze will:

- Compute the energy values
- Compute the total energy
- Compute the average energy
- Mark the frame as valid
- Return if the frame is valid

Computing and Summing Energy

When computing the energy values, be sure not to overrun either array. Use the smaller of the count value in the frame and 16 (the frame array size). You are allowed to use the constant $16 in your code. For our purposes, energy is exactly the square of the amplitude. (In real physics, energy is directly proportional to amplitude.)

Be very careful about type (length, really). The PCM data is 16 bits. The energy value needs 32 bits to hold the square of the PCM data. The sum of the total energy needs 64 bits. We know from lab 5 that the 16 bit PCM data is roughly in the range of +/- 32K. A glance at the sample output shows that the energy levels can hit values on the order of 1B, far too big for a C short. Another glance at the sample output shows total energy hitting 6B, which is too big for a C int.

Average Energy and Using Divide

To compute the average energy, we need to divide.

Because the original 8086 chip had a severely limited number of transistors, divide has very particular idiosyncrasies. The thing you will divide starts out in RAX or a smaller version of it. Then we do magic on it to extend it, which will trash RDX (or a smaller subset of it). Then we can divide by any of the other registers. The result winds up in RAX or a smaller subset of it.

The example code that follows assumes that we are working with assembler longs (C ints). This means that EAX holds the value we want divided. If ECX holds the value we are dividing by, we would get the code below. It can be in any of the caller saved other than EAX or EDX. That would yield:

```
cltd       # the magic conversion that trashes edx
idivl %ecx # divide eax by ecx, result is in eax
```

If your code needs different sizes than the example above, be sure to convert that code to the sizes you need, which will mean a change to the convert opcode. (The convert opcode are listed in slide deck 30, Favorite Opcodes.) In the example above, the opcode reads "**c**onvert **l**ong **t**o **d**ouble long."

Once you compute the average energy, be sure to fill in the value into the frame structure.

Valid Frame

The frame is valid if the average energy is at or above the lower threshold and also at or below the upper threshold. Mark the frame data with that result and be sure to return it as the return value of the function.

Memory Touches

The function can only access any given memory location **one time**. Use registers to do any computations. For example. do not use the struct total_energy member to sum total energy, use a register and write the register to the struct when the loop completes.

- You can read the count member one time
- You can read pcm values one time each
- You can write energy values one time each
- You can write total energy value one time
- You can write average energy value one time

- You can write the frame valid member one time

If you run out of caller saved registers, you may resort to using callee saved. Do not do so until all of the caller saved registers are put to use.

Plan of attack

Build a little, test a little! As soon as you get the first step below working, create a zip file that you could turn in. Make the zip again when you complete the next step. Always have a zip file of working code ready to turn in, even if it is incomplete.

Version 1 of analyze should only loop through the data and compute and fill in the energy values. Use atest to make sure that the energy values are correct and that your code doesn't blow up. Don't bother computing anything else or doing anything else. Make sure that your loop terminates correctly.

Version 2 sums the energy values and writes that sum to the struct. Use atest to make sure it is correct.

Version 3 does the divide to compute average energy and fills in the average value to the struct. Again, use atest to make sure it is right.

Version 4 does the comparisons on average energy to compute validity. It writes that to the struct and returns that value. At this point, atest should show that your analyze is complete.

The search function

```
struct Frame *search(int count, struct Dataset *data_ptrs[]);
```

The search function goes through datasets using a_shim to fill out data about the frame that each dataset carries. Search should return the frame pointer to the valid frame that has the highest average energy. If no frames are valid, search returns NULL. At the end, search will print either a message that states that no valid frames were found or it will print the lowest and highest average energy values that it encountered in the valid frames. Search should print once per frame whether it was valid or invalid and the number of samples in that frame.

Search should be mostly confined to registers. A mix of calle and caller saved registers might be best. Search should minimize the number of memory touches it does.

- You can read a data set pointer from the array of dataset pointers once per loop
- You can read from any single location in the frame data exactly one time

If you run out of callee saved registers, allocate space in the stack for variables that simply must survive a function call. Be sure to maintain stack alignment if you do. A search function that uses stack variables does not qualify for the bonus points for search.

Search's Task List:

- Loop through the datasets and call a_shim on each one
- Print if a dataset is valid or invalid and the number of samples it has (delete this if you do the 2 point bonus)
- Find the valid dataset with the highest average energy
- Track the highest average energy of all valid datasets

- Track the lowest average energy of all valid datasets
- Print the high and low (if there were valid datasets) or print that there were no valid ones
- Return NULL or the frame pointer of valid dataset with the highest average energy

Plan of Attack for Search

Work out the overall design before you write any code! Search, as a function, does too much. In C code, it doesn't meet the ten-line limit. Design is key here. What changes in how you think of search if you break it into three parts? Those would be the first two task in the first part, the middle 3 tasks in the second part, and the last two tasks in the third part. What assumptions change when you organize it that way?

As with analyze, you should implement this function is stages. Once a stage works, rebuild your zip file. That way you always have something to turn in that works.

Shims

The driver code deals with the call to s_shim. Your code has to call **print** and not printf. Search **must** call a_shim and not analyze. Your code must make all calls with the stack aligned on a 16 byte boundary.

Output

Your output should be similar to these. The entire output is also available on Piazza.

atest

```
[kirby.249@coe-dnc268474s instructor]$ ./atest

Running analyze with range 50 to 80 with 1 samples
1 In range (64).  Total energy was 64
This frame has
    valid = true
    1 samples
    64 total energy
    64 avg energy
       PCM          Energy
         8              64

Running analyze with range 100 to 1000 with 24 samples
0 NOT in range (4).  Total energy was 64
This frame has
    valid = false
    24 samples
    64 total energy
    4 avg energy
       PCM          Energy
         2               4
        -2               4
         2               4
```

```
        -2              4
         2              4
        -2              4
         2              4
        -2              4
         2              4
        -2              4
         2              4
        -2              4
         2              4
        -2              4
         2              4
        -2              4
         2
        -2
         2
        -2
         2
        -2
         2
        -2

Running analyze with range 1047552 to 1049600 with 5 samples
1 In range (1048576).  Total energy was 5242880
This frame has
    valid = true
    5 samples
    5242880 total energy
    1048576 avg energy
       PCM          Energy
      1024         1048576
     -1024         1048576
      1024         1048576
     -1024         1048576
      1024         1048576

Running analyze with range 10 to 20 with 34 samples
0 NOT in range (378003493).  Total energy was 6048055894
This frame has
    valid = false
    34 samples
    6048055894 total energy
    378003493 avg energy
       PCM          Energy
         2               4
        -4              16
         8              64
       -16             256
      2048         4194304
     -8192        67108864
     16384       268435456
    -32768      1073741824
```

```
     32767      1073676289
    -32768      1073741824
     32767      1073676289
    -32768      1073741824
    -16384       268435456
      8192        67108864
     -2048         4194304
        16             256
        -8
         4
        -2
         1
         0
        -1
         0
         2
        -2
         4
         0
        -4
         3
         0
        -3
         1
         0
        -1
[kirby.249@coe-dnc268474s instructor]$
```

lab6

```
[kirby.249@coe-dnc268478s instructor]$ ./lab6

Lab 6 calling s_shim with 4 frames
search found a valid frame with 1 samples
search found an invalid frame with 4 samples.
search found a valid frame with 5 samples
search found an invalid frame with 24 samples.
search: The high frame has 1048576 avg energy
    We saw valid frames as low as 64 avg energy
Lab6: s_shim returned this frame:
This frame has
    valid = true
    5 samples
    5242880 total energy
    1048576 avg energy
       PCM          Energy
      1024         1048576
     -1024         1048576
      1024         1048576
     -1024         1048576
      1024         1048576
```

```
Lab 6 calling s_shim with 0 frames
search: No valid frames found
Lab6: s_shim did not find any valid frames

Lab 6 calling s_shim with 2 frames
search found an invalid frame with 4 samples.
search found an invalid frame with 24 samples.
search: No valid frames found
Lab6: s_shim did not find any valid frames
[kirby.249@coe-dnc268478s instructor]$
```

Bonus

Analyze: fits in the caller saved registers without resorting to using any callee saved registers or variables in memory: +1

Search:

> You only need 4 callee saved registers for +1 point
> -or-
> Delete the two "Search found…" printf calls and use only 3 callee saved registers for +2 points.

Register Allocation

Register allocation policy is graded.

Some of the required functions are leaf-level and the some are 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. 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.

Near the top of every function, create in comments your pocket guide to the registers. Each register used in the function **must** have a comment telling what that register means. For example:

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

In your comments you should refer to the registers not by name but by what they mean.

Comments & Labels

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.
```

When setting parameters for print, you comment for rdi should be the double quoted string. Do the first parameter first so that you can then use the comment to guide what goes in other registers.

```
# set up the next print call:
movq $input_print, %rdi    #     "%d was passed in to analyze"
# then do the move to rsi and later parameters
```

Use whitespace and comments to break you code up into sections. The labels will help, but put comments in at the beginning of a chunk of related lines of code. The above code would have a blank line prior to the bold comment about setting up print. After the call to print would be another blank line and then a comment about what the next group of statements do.

Use meaningful names on the **labels** while you are at it. Do **not** use machine-generated labels like .LC0 in your code. The above code shows a label of "input_print" which is far more informative than .LC0. So your labels in the data and rodata section need to be meaningful as well as the labels that are jump targets.

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
- If it qualifies for bonus points
- 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 needed to make all targets build

Be sure that the zip target in the makefile self-tests by making atest and lab6 (and possibly bonus) 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.

Machine Generated Code

Do not use –S with gcc to generate any assembler from C code. Do not use any AI such as github copilot to do this lab. Doing so is academic conduct.