# CprE 381 - Computer Architecture and Assembly Level Programming Fall 2017 Lab-3

**INTRODUCTION:**  
Last week, we saw how to use SimpleScalar and simulate programs on it, tweak some configurations on the simulator, read and analyze certain performance metrics. In the first part of this lab, you are going to continue using SimpleScalar to verify Amdahl’s Law. The second part is aimed at exposing you to Assembly programming. You are going to see some programs in assembly and try to make sense of it.

**Part A (Amdahl’s Law):**In this section, we are going to verify Amdahl’s law (refer to your textbook/lecture slides for description and formulae related to it).

Download the zip file, extract and place it in **Drive(X:) > cpre381 > Lab3**.  
(Hit Windows+r, and type [\\dfs-2.ece.iastate.edu\home\username](file:///\\dfs-2.ece.iastate.edu\home\username) to connect X: from Windows)  
(Create a **Lab3** folder inside **cpre381** folder if you haven’t already)

The files given to you contain programs that perform division operations. Multiplication/division operations tend to take multiple cycles to execute. Our aim is to improve the overall performance of the program by making the division operation faster. This can be achieved by replacing the regular division operation with shift operation. To divide by any value of 2 to the power of N (i.e. 2^N), shift the bits N times to the right.   
  
**Strategy:**  
  
We are going to assess the speedup in terms of number of cycles (since the clock is not being changed, execution time is proportional to cycle count). For that, we need the number of cycles of the entire program and that of the division operation(s) alone, since we plan to achieve speedup by making the division faster. When you compile a program with gcc, it adds a significant amount of overhead to your program; hence the cycle count that your simulator will give you cannot be used directly for your calculations. Here is what you will do:

(a)Write a program that performs division operation, something like  
   
 x = 7; y = x/3; z = x/y

Compile w/o optimization (-O0) and simulate using **sim-outorder.** Note the #cycles.

15722

**Remember to generate and edit the config file and make the in-order pipeline to true like you did in last lab. Pass it as argument when running the simulator. Do this for all experiments in this lab.**

(b) Write an identical program but without the division operation (basically this will contain all headers and variable initialization identical to your first program but the program will not do anything).

Compile w/o optimization (-O0) and simulate using **sim-outorder.** Note the #cycles.

15633

From the # cycle values in (a) and (b) find out the #cycles required for the division operation.

15722 – 15633 = 89 / 2 = 44.5   
  
(c)Compare **opt\_division.c** with **unopt\_division.c** and report differences in the code**.**

**The opt uses shifting of bits and unopt uses division.**

(d). Using Amdahl’s Law (refer lecture slides if necessary)and your result from (b),manually calculate the expected speedup going from **unopt\_division.c**to **opt\_division.c** assuming that the division operation achieved using shift operation brings down the cycle count to 1. Report your speedup.

3000 clock count – opt and unopt

Unopt is going to take 3000 \* 44.5 more clock cycles

So SU = 133500/3000 = 44.5

(e). Run the precompiled binaries **unopt\_division** and **opt\_division**on **sim-outorder** and find any changes in cycle count and CPI of the programs. Verify this with the manually calculated speedup from part (d). (Do not panic if your numbers are an order of magnitude higher than your manual calculation, we do not expect to see an exact match in speedup).

Unopt - 3.0060 CPI and 118271 total of cycles

Opt - 1.6752 CPI and 42803 total of cycles

118271/42803 = 2.763 SU

**Part B (Assembly Program):**  
In this section we will generate our own assembly code from a high level C code. SimpleScalar implements a MIPS like instruction set called Portable Instruction Set Architecture (PISA). Most of the primitive instructions in PISA such as load/store, addition, multiplication are very much identical to MIPS.

The assemblies for **opt\_division.c** and **unopt\_division.c** have been provided to you (with “***.s”*** extensions).

**Side note**:   
If you would like to compile your own C programs to human-readable assembly you can use the “-S” flag in gcc. This prevents gcc from creating the binary and instead creates the assembly “.s” file.  
To read assembly files, ignore all overhead at the beginning of the file. Your main function should start after the ***main:*** in the file.

(f). Identify and report differences between the optimized and unoptimized assemblies in terms of instructions (sra).

Unopt – DIV - 3000  
 There are about 18 more instructions in the unopt and there are no div commands in opt  
(g). Now run **unopt\_division** binary on **sim-profile** using the **–iprof** flag. Do the same for **opt\_division** binary provided to you in the zip file. Compare instruction mix (div vs. sra).

Unopt – DIV – 3000 , sra – 12

Opt – Div – 0 sra - 3009

(h) Annotate the given ***opt\_division.s*** assembly code (starting from main: to .end main), line-by-line, and explain which line in the ***opt\_division.c*** code corresponds to which line(s) in the assembly code.

(i) Repeat (h) for ***unopt\_division.s*** and ***unopt\_division.c***.

**Submission**All submissions are through Blackboard. Write your answers in the attached report file. Submit your modified file(s) (config\_out.txt) as well, along with your report.