In [50]:
```
# This cell runs automatically.  Don't edit it.
from notebook import *
```

Double Click to edit and enter your
1. Name
2. Student ID
3. @ucsd.edu email address

Lab 1: The Performance Equation

FOR THIS LAB ONLY, I'M PROVIDING A PDF OF THE LAB AHEAD OF TIME.

Welcome to the first lab of CSE142L!

The main goals of this lab are:

1. To get you set up the lab environment
   A. Github
   B. Github Classroom
   C. Docker
   D. UCSD's Datahub cluster
   E. Jupyter Notebook
2. Gain experience with the performance equation
3. Collect a list of **Interesting Questions** that you probably cannot answer now, but will be able to by the end of the course.

This lab will be completed on your own.

Check Gradescope for due date(s).

1 FAQ and Updates

- There are no updates, yet.

2 Pre-Lab Reading Quiz

Part of this lab is pre-lab quiz. It's on Gradescope, and it's due **before class on the day the lab is assigned**. It's not hard, but it does require you to read over the lab before class.

3 Browser Compatibility

We are still working out some bugs in some browsers. Here's the current status:

1. Chrome -- well tested. Preferred option.
2. Firefox -- seems ok, but not thoroughly tested.
3. Edge -- seems ok, but not thoroughly tested.
4. Safari -- not supported at the moment.
5. Internet Explorer -- not supported at the moment.

At the moment, the authentication step must be done in Chrome.

4 About Labs In This Class

This section is the same in all the labs. It's repeated here for your reference.

Labs are a way to **learn by doing**. This means you *must* **do**. I have built these labs as Jupyter notebooks so that the "doing" is as easy and seamless as possible.

In this lab, what you'll do is answer questions about how a program will run and then compare what really happened to your predictions. Engaging with this process is how you'll learn. The questions that the lab asks are there for several purposes:

1. To draw your attention to specific aspects of an experiment or of some results.
2. To push you to engage with the material more deeply by thinking about it.
3. To make you commit to a prediction so you can wonder why your prediction was wrong or be proud that you got it right.
4. To provide some practice with skills/concepts you're learning in this course.
5. To test your knowledge about what you've learned.

The questions are graded in one of three ways:

1. "Correctness" questions require you to answer the question and get the correct answer to get full credit.
2. "Completeness" questions require you to answer the question.
3. "Optional" questions are...optional. They are there if you want to go further with the material.

Some of the "Completeness" problems include a solution that will be hidden until you click "Show Solution". To get the most from them, try them on your own first.

Many of the "Completeness" questions ask you to make predictions about the outcome of an experiment and write down those predictions. To maximize your learning, think carefully about your prediction and commit to it. **You will never be penalized for making an incorrect prediction.**

You are free to discuss "Completeness" and "Optional" questions with your classmates. You must complete "Correctness" questions on your own.

If you have questions about any kind of question, please ask during office hours or during class.

4.1 How To Succeed On the Labs

Here are some simple tips that will help you do well on this lab:

1. Read/skim through the entire lab *before* class. If something confuses you, you can ask about it.
2. Start early. Getting answers on edstem/piazza can take time. So think through the lab questions (and your questions about them) carefully.
   A. Go through the lab once (several days before the deadline), do the parts that are easy/make sense
   B. Ask questions/think about the rest
   C. Come back and do the rest.
3. Start early. The DSMLP cluster gets busy and slow near deadlines. "The cluster was slow the night of the deadline" is not an excuse for not getting the lab done and it is not justification for asking for an extension.
4. Follow the guidelines below for asking answerable questions on edstem/piazza.

You may think to yourself: "If I start early enough to account for all that, I'd have to start right after the lab was assigned!" Good thought!

The Cluster Will Get Slow DSMLP and our cloud machines will get crowded and slow *before every deadline*. This is completely predictable. DSMLP can also get crowded due to deadlines in other courses. You need to start early so you can avoid/work around these slowdowns. Unless there's some kind of complete outage, we will not grant extensions because the servers are crowded.

4.2 Getting Help

You might run into trouble while doing this lab. Here's how to get help:

1. Re-read the instructions and make sure you've followed them.
2. Try saving and reloading the notebook.
3. If it says you are not authenticated, go to the the login section of the lab and (re)authenticate.
4. If you get a `FileNotFoundError` make sure you've run all the code cells above your current point in the lab.

5. If you get an exception or stack dump, check that you didn't accidentally modify the contents of one of the python cells.
6. If all else fails, post a question to edstem/piazza.

4.3 Posting Answerable Questions on Edstem/Piazza

If you want useful answers on edstem/piazza, you need to provide information that is specific enough for us to provide a useful answer. Here's what we need:

1. Which part of which lab are you working on (use the section numbers)?
2. Which problem (copy and paste the *text* of the question along with the number).

If it's question about instructions:

1. Try to be as specific as you can about what is confusing or what you don't understand (e.g., "I'm not sure if I should do *X* or *Y*.")

If it's a question about an error while running code, then we need:

1. If you've committed anything, your github repo url.
2. If you've submitted a job with `cse142` you *must* provide the job id. It looks like this: `544e0cf2-4771-43c3-86f8-1c30d7af601f` . With the id, we can figure out just about anything about your job. Without it, we know nothing.
3. The *entire* output you received. There's no limit on how long an edstem/piazza post can be. Give us all the information, not just the last few lines. We like to scroll!

For all of the above **paste the text** into the edstem/piazza question. Please **do not provide screen captures**. The course staff refuses to type in job ids found in screen shots.

> **We Can't Answer Unanswerable Questions** If you don't follow these guidelines (especially about the github repo and the job id), we will probably not be able to answer your question on edstem/piazza. We will archive it and ask you to re-post your question with the information we need.

4.4 Keeping Your Lab Up-to-Date

Occasionally, there will be changes made to the base repository after the assignment is released. This may include bug fixes and updates to this document. We'll post on piazza/edstem when an update is available.

In those cases, you can use the following commands to pull the changes from upstream and merge them into your code. You'll need to do this at a shell. It won't work properly in the notebook. Save your notebook in the browser first.

```
cd <your directory for this lab>git remote add upstream $(cat .star
ter_repo)  # You need to do this once each time you checkout a new
```

```
 lab. It will fail
                                                    # harmlessly if you r
un it more than once.
cp Lab.ipynb Lab.backup.ipynb                       # Backup your work.
git commit -am "My progress so far."                # commit your work.
git pull upstream main --allow-unrelated-histories -X theirs # pull
the updates
```

Or you can use the script we provide:

```
./pull-updates
```

Then, reload this page in your browser.

4.5 How To Use This Document

You will use Jupyter Notebook to complete this lab. You should be able to do much of this lab without leaving Jupyter Notebook. The main exception will be some of the programming assignments. The instructions will make it clear when you should use the terminal.

4.5.1 Logging In

If you haven't already, you can go to the login section of the lab and follow the instructions to login into the course infrastructure.

4.5.2 Running Code

Jupyter Notebooks are made up of "cells". Some have Markdown-formatted text in them (like this one). Some have Python code (like the one below).

For code cells, you press `shift-return` to execute the code. Try it below:

In [2]:
```
print("I'm in python")
```

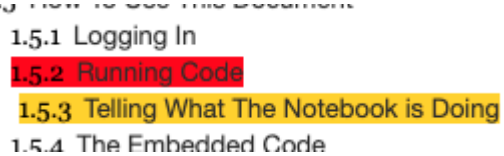
Code cells can also execute shell commands using the `!` operator. Try it below:

In [6]:
```
!echo "I'm in a shell"
```

4.5.3 Telling What The Notebook is Doing

The notebook will only run one cell at a time, so if you press `shift-return` several times, the cells will wait for one another. You can tell that a cell is waiting if it there's a `*` in the `[]` to the left the cell:

You'll can also tell *where* the notebook is executing by looking at the table of contents on the left. The section with the currently-executing cell will be red:



4.5.4 What to Do Jupyter Notebook It Gets Stuck

First, check if it's actually stuck: Some of the cells take a while, but they will usually provide some visual sign of progress. If *nothing* is happening for more than 10 seconds, it's probably stuck.

To get it unstuck, you stop execution of the current cell with the "interrupt button":

You can also restart the underlying python instance (i.e., the confusingly-named "kernel" which is not the same thing as the operating system kernel) with the restart button:

Once you do this, all the variables defined by earlier cells are gone, so you may get some errors. You may need to re-run the cells in the current section to get things to work again.

You can also reloading the web page. That will leave Python kernel intact, but it can help with some problems.

4.5.5 Common Errors and Non-Errors

1. If you get `sh: 0: getcwd() failed: no such file or directory`, restart the kernel.
2. If you get `INFO:MainThread:numexpr.utils:Note: NumExpr detected 40 cores but "NUMEXPR_MAX_THREADS" not set, so enforcing safe limit of 8.`. It's not a real error. Ignore it.
3. If you get a prompt asking `Do you want to cancel them and run this job?` but you can't reply because you can't type into an output cell in Jupyter notebook, replace `cse142 job run` with `cse142 job run --force`. (see useful tip below.)
4. If you get an "internal server error", trying running the job again.

4.5.6 Useful Tips

1. If you need to edit a cell, but you can't you can unlock it by pressing this button in the tool bar (although you probably shouldn't do this because it might make the lab work incorrectly. A better choice is to copy and paste the cell, *and then* unlock the copy):

4.5.7 The Embedded Code

The code embedded in the lab falls into two categories:

1. Code you need to edit and understand.
2. Code that you do not need to edit or understand -- it's just there to display something for you.

For code in the first category, the lab will make it clear that you need to study, modify, and/or run the code. If we don't explicitly ask you to do something, you don't need to.

Most of the code in the second category is for drawing graphs. You can just run it with shift-return to the see the results. If you are curious, it's mostly written with `Pandas` and `matplotlib`. The code is all in `notebook.py`. These cells should be un-editable. However, if you want to experiment with them, you can copy *the contents* of the cell into a new cell and do whatever you want (If you copy the cell, the copy will also be uneditable).

> **Most Cells are Immutable** Many of the cells of this notebook are uneditable. The only ones you should edit are some of the code cells and the text cells with questions in them.

> **Pro Tip** The "carrot" icon in the lower right (shown below) will open a scratch pad area. It can be a useful place to do math (or whatever else you want.

4.5.8 Showing Your Work

Several questions ask you to show your work for calculations. We don't need anything fancy. Many of the questions ask you to compute something based on results of an experiment. Your experimental results will be different than others', so your answer will be different as well.

To make it possible to grade your work (and give you partial credit), we need to know where your answer came from. This why you need to show your work. For instance this would be fine as answer to "On average, how many weeks do you have per lab?":

```
Weeks in quarter/# of labs = 10/5 = 2 weeks/lab
```

2 significant figures is sufficient in all cases, but you can include more, if you want.

If you are feeling fancy, you can use LaTex, but it's not at all required.

When it's appropriate, you can also paste in images. However, Jupyter Notebook is flaky about it. Save your notebook by clicking the disk icon:

4.5.9 Answering Questions

Throughout this document, you'll see some questions (like the one below). You can double click on them to edit them and fill in your answer. Try not to mess up the formatting (so it's easy for us to grade), but at least make sure your answer shows up clearly. When you are done editing, you can `shift-return` to make it pretty again.

A few tips, pointers, and caveats for answering questions:

1. The answers are all in github-flavored markdown (https://guides.github.com/features/mastering-markdown/) with some html sprinkled in. Leave the html alone.
2. Many answers require you to fill in a table, and many of the | characters will be missing. You'll need to add them back.
3. The HTML needs to start at the beginning of a line. If there are spaces before a tag, it won't render properly. If you accidentally add white space at the beginning of a line with an html tag on it, you'll need to fix it.
4. Text answers also need to start at the beginning of a line, otherwise they will be rendered as code.
5. Press `shift-return` or `option-return` to render the cell and make sure it looks good.
6. There needs to be a blank line between html tags and markdown. Otherwise, the markdown formatting will not appear correctly.

You'll notice that there are three kinds of questions: "Correctness", "Completeness", and "Optional". You need to provide an answer to the "Completeness" questions, but you won't be graded on its correctness. You'll need to answer "Correctness" questions correctly to get credit. The "Optional" questions are optional.

In [ ]:

Give it a try:

Question 1 (Completeness)

What will you do with your Jupyter Notebook to turn it in (Delete *all* incorrect answers)? Then fill in the table. Fix the formatting of the last line and the closing tag.

1. Print it out on paper and slide it under the professor's door. 2. Follow the directions at the end of the lab to produce a pdf 3. Read it aloud to a TA during office hours 4. Submit via gradescope.

Random Fact	Value	Second choice
Your favorite color		
Favorite food on campus		
Time of day		

```
This answer is formatted poorly.
</div>
```

Show Solution

5 Logging In To the Course Tools

In the course you will use some specialized tools to let you perform detailed measurements of program behavior. To use them you need to login with your `@ucsd.edu` email address using the instructions below. **You need to use the email address that appears on the course roster. That's the email address we created an account for. In almost all cases, this is your `@ucsd.edu` email address.**

You'll do this periodically when you get an error about not being authenticated. You can return to this notebook (or any other of the lab notebooks) to login at any time.

Here's what to do:

1. Enter your `@ucsd.edu` email address in quotes after `login` below. It'll take a few seconds to load.
2. Click the google "G" login button below and login with your `@ucsd.edu` email address.
3. **Click the google button regardless of whether it says "sign in" or "signed in". Then be sure to select your `@ucsd.edu account` if it shows you multiple google acocunts**
4. You'll see a very long string numbers an letters appear above. Click "Copy it" to copy it.

Note: If it doesn't give you a choice about which account to log into and authentication fails, that means you are logged into a single Google account and that account is *not* your `@ucsd.edu` account. You'll have to log into your `@ucsd.edu` through Gmail or through Chrome's account manager and then try again.

Use Chrome The login process doesn't seem to work properly with Safari or Firefox. Use Chrome to login. You can use any of the other compatible browsers you want for the doing the rest of the lab, and it should be fine.

In [2]:
```
login("<Your @ucsd.edu email address>")
```

Next step: Paste it below between the quote marks. Press `shift-return`.

In [1]:
```
token("your_token")
```

It should have replied with

```
You are authenticated as <your email>
```

You are now logged in! Try submitting a job:

In [1]:
```
!cse142 job run "echo Hello World"
```

If you see "Hello World", you're all set. Proceed with the lab!

> Delete your token from the above cell. Because your token is esssentially your username and password combined, you should treat it like a password or ssh private key. **Sharing your token with another student or posessing another student's token is an AI violation**.

6 Grading

This is a 2 week lab.

Your grade for this lab will be based on your completion and submission of this notebook.

Part	value
Reading quiz	3%
Jupyter Notebook	90%
Programming Assignment	5%
Post-lab survey	2%

We will grade 5 of the "completeness" problems. They are worth 3 points each. We will grade all of the "correctness" questions.

Check Gradescope for the due dates.

Instructions for submitting the lab are at the end of the lab.

No late work or extensions will be allowed.

7 Academic Integrity Agreement

To continue in the class, you need to agree to the following:

At UCSD, academic integrity[1] means that you have the courage, even when it is difficult, to only submit academic work that is honest, responsible, respectful, fair, and trustworthy. When you excel with integrity in computer science, it means that you:

Honest submit work that is a truthful demonstration of your knowledge and abilities (rather than the knowledge and abilities of another)

Responsible manage your time so that you are not pressured to complete an assignment at the last minute

Respectful acknowledge the contributions of others to your work by citing them when ve used their words or ideas (e.g., after I've spoken to classmates or after I've used portions of a code written by another if permitted)

Fair complete your academic work according to stated standards and expectations even when it takes longer or re struggling

Trustworthy can be trusted to be honest, responsible, respectful, and fair even when no one is watching you.

When you act contrary to these values, you are cheating. Cheating undermines trust between students and professors, the value of the UCSD degree, and your learning/development of skills.

While we can't list every behavior that would be cheating, we can give you some illustrative examples like the following:

- Submitting any source code written by another person or copied from another person, submitting homework answers which were produced by another student.
- Submitting code/homework you have previously submitted to another course for credit without first obtaining permission from the instructor. The same restriction holds for publicly available code/homework solutions that you haven't written. Taking notes taken during any discussions with classmates about an assignment is prohibited.
- Using words or text written by someone else without citing text appropriately. Every figure or sentence fragment must be appropriately decorated with quotation marks or indention to indicate very clearly that someone else wrote the text. In addition, the passage must be labeled with a citation or citation number which refers to a footnote or bibliographic entry. Citing a paper once is not enough. Remember: citations should be used to illuminate a viewpoint which you hold. They are not a substitute for expressing your own ideas in your own words.
- Submitting any portion(s) of an assignment you have previously submitted for credit in another course.
- Copying from a neighbor during an exam or using an unauthorized aid to help you on your exam.
- Altering a graded exam or assignment and resubmitting it for regrade
- Allowing someone else to complete an assignment or exam for you, or allowing them to pretend to be you in class (e.g., by signing an attendance form or clicking for you).

- Making available to others source code, documentation, or notes useful for completing an assignment. You should neither produce, procure, nor accept such material. This includes students in current, past, and future offerings of the course, and applies to electronic transmissions including email, web pages, ftp, and so on, as well as hard copy such as source code listings.
- Possessing (at any time) source code, data, or answers to homeworks or lab questions created by another student. Having had any of these items in your possession (e.g., in your directory on the campus servers or on your personal computer) at any time, constitutes cheating. You should never accept these materials from anyone for any reason.
- Running other students' code from your account or allowing another student submit code using your credentials.

If the behavior you are considering isn't listed here, don't assume that it is allowed. Rather, you should always do independent work unless told otherwise. And before completing your academic work in a certain way, you should ask is it honest, respectful, responsible, fair, and trustworthy. You can also ask yourself "Would I be okay if my methods were exposed to the TA, professors, and fellow students?" If the answer is no, you shouldn't do it.

If you have any questions about what is and isn't cheating, be sure to discuss them with the instructor.

Any student who cheats, thereby undermining integrity, will be reported to the Academic Integrity Office. Students who cheat face various disciplinary sanctions as well as academic penalty imposed by the instructor in the course. *Academic penalties include, but are not limited to, receiving a grade of 0 for the assignment or test in question, and receiving an 'F' for the course.*

[1] For more information on academic integrity, including how you can excel with integrity, as well as information on sanctioning guidelines for cheating, visit the Academic Integrity Office website at: http://academicintegrity.ucsd.edu (http://academicintegrity.ucsd.edu)

Question 2 (Correctness - 1pts)

Please affirm your adherence to this agreement

Type 'I excel with integrity' here: [`type it (leave the brackets)`]

By submitting this file, I, [`Your Name`], a student enrolled in CSE142L affirm the principle of academic integrity and commit to excel with integrity by completing all academic assignments in the manner expected as described above, informing the instructor of suspected instances of academic misconduct by my peers, and fully engaging in the class and its related assignments for the purpose of learning.

To electronically sign this document, Enter your full name, date, and student ID below:

[`full name`]/[`date`]/[`Student ID #`]

This document was written in part by Rick Ord, CSE Lecturer and Dr. Bertram Gallant, Director of the UCSD Academic Integrity Office.

8 Skills to Learn

1. Get access to docker
2. git/GitHub basics
3. Navigating a Jupyter Notebook
4. Think code and predict its behavior.

9 Building and Running Code

In this class, we will spend a lot of time measuring the behavior of programs. To do this accurately, we need to run the code by itself on a machine, so that other programs don't interfere with our measurements. As you'll experience, DSMLP is a very "noisy" environment from a performance perspective.

To get good measurements, we'll run our experiments "in the cloud" on some dedicated *bare metal* servers where nothing else runs. Since those servers are expensive, we have to share 12 of them across all the students in the course. So, rather than have you log in directly, you can build your code on DSMLP, debug your code on DSMLP, and then submit "jobs" to run in the cloud.

9.1 Build The Code Locally

We'll build executables using `make`. For instance, you can compile `hello_world.cpp` into `hello_world.exe` by typing this at your Linux shell prompt:

```
make hello_world.exe
```

Or, you can do that from right here in the note book by putting a `!` in front. Type

```
!make hello_world.exe
```

Like this:

In [49]:
```
!make hello_world.exe
```

```
make: 'hello_world.exe' is up to date.
```

9.2 Run the Code Locally

Then you can run it like so:

In [ ]:
```
!./hello_world.exe
!./hello_world.exe something
```

Question 3 (Completeness)

In the code cell below use the `!` to build and run `microbench.exe`. (`microbench.exe` will say "Execution Complete" and exit).

In [ ]:
```
#Put your commands here
```

9.3 Run the Code Remotely

To run a job, you tell `cse142 job run` what command you would like to run. It gathers up *all the files* (well, not quite all of them, but most of them) in your lab directory, ships them to the bare-metal server, unpacks them, runs your command, gathers up the files that have changed, ships them back and unpacks them in your directory. This makes it look like the command ran locally.

Let's try it with something simple:

In [25]:
```
!make hello_world.exe
!cse142 job  run './hello_world.exe'
```

Note that we didn't need to compile `hello_world.exe` on the remote machine because `cse142` copied the executable to the cloud.

The output shows some job status information as it runs. The job moves through several states as it executes. The time it spends `RUNNING` is time it took your code to execute. `PUBLISHED` and `SCHEDULED` (maybe not shown above) means it was waiting for a machine.

And the output of the job is near the end:

```
Hello cse142L!
```

You can run anything you want on the other side. For fun, let's build the executable over there and then run it:

In [33]:
```
!make clean
!cse142 job run --force 'make hello_world.exe; ./hello_world.exe'
```

Now you can see the compiler output and you'll notice that it copied back the files it modified while building `hello_world.exe`.

```
./build/hello_world.cpp
./build/hello_world.d
./build/hello_world.o
./hello_world.exe
```

With great power comes great responsibility. You can run pretty much whatever you want on the remote machine, but the code runs in a docker-based sandbox (using the same image you are running locally) and there's a time limit. Nevertheless, there are probably ways for you to crash the remote machine or cause it to malfunction. Doing so intentionally is against the rules. If we find you've done so, we'll disable your account and you'll have to find some other way to complete the labs.

10 Analyzing Program Behavior with the Performance Equation

Keep track of questions you have. This lab is more about collecting questions than finding answers. I've called out some interesting questions throughout the lab. The last question of the lab asks for *other* questions you had while examining the data you'll collect below, so keep track of them as you work through the lab.

10.1 Meet the Code!

In this lab we are going to analyze a few simple funtions in `microbench.cpp` . The first is `baseline_int()` :

In [51]:
```
render_code("microbench.cpp", show="baseline_int")
```

```
// microbench.cpp:17-29 (13 lines)
extern "C" uint64_t *__attribute__((noinline)) baseline_int(uint64_t * ar
ray, unsigned long int size) {
        //uint64_t * array = new uint64_t[size];
        for(uint i = 0; i < size; i++) {
                array[i] = 0;
        }

        for (uint j = 0; j < 3; j++) {
                for(uint i= 1 ; i < size; i++) {
                        array[i] += i/(1+j)+array[i - 1];
                }
        }
        return array;
}
```

`baseline_int()` initilizes `array` and then does some multiplies and additions to update it's contents. It's not a useful computation, so don't spend time trying to figure out what it does.

Compile the code to create `microbench.exe` and then run it:

In [ ]:
```
!make microbench.exe
```

In [ ]:
```
!./microbench.exe --help
```

As you can see, there are quite a few command line options for such a simple program. These options are the interface to the data collection library we will be using in the class. They control how our "functions under test" (FUTs) (i.e., `baseline_int()`) will be run. We'll learn about them in more details as the quarter progresses.

For now, let's use three arguments:

- `--function` to run `baseline_int`
- `--stats` to put the results in `first.csv`
- `--reps` will run the test 2 times
- `--size` will run tests on two different sized arrays.

`microbenchmark.exe` will run all the funtions listed after `--function` for all the array sizes given by `--size`, and it will do it all 2 times, so that's a total of 4 measurements it will take.

We can run it locally:

In [ ]:
```
!./microbench.exe --stats first.csv --reps 2 --size 1024 2048 --function l
```

`first.csv` contains some data about how the program run. You can see the raw data:

In [ ]:
```
!cat first.csv
```

And render it nicely:

In [ ]:
```
render_csv("first.csv")
```

Here's what the column means:

1. `size` is the size of the `array`
2. `cmdlineMHz` is the CPU clock speed it tried to run at. '0' means the default.
3. `function` is the the function in this line is for.
4. `rep` is the 'repetition number' (explained below).
5. `WallTime` is the number of seconds the program ran.
6. Ignore `Unnamed: 6` its existence is a bug I haven't had time to fix.

Let's focus on a few columns:

In [ ]:
```
render_csv("first.csv", columns=["function", "rep", "size"])
```

10.2 Measuring The Performance Equation

Now that we know how to take measurements, we can try to understand `baseline_int()` 's performance. We will do this using the performance equation:

```
ET = IC * CPI * CT
```

So, we'll need to measure `IC` (instruction count), `CPI` (cycles per instruction), `CT` (cycle time), and `ET` , and we'll do that using "performance counters". `microbench.exe` already has support for performance counters built in, we just need to tell it to collect data. We can do that using the command below. Here's what the additions to the command line mean.

1. `--reps` will run the test 25 times so we can average across lots of runs.
2. `--stat-set ./PE.cfg` configures `microbench.exe` to collect the data we need. (If your curious, take a look at `PE.cfg` in the lab directory)
3. We set the clock speed to 3500MHz with `--MHz 3500` .

Performance counters *only* work on the bare metal servers. In fact, `--stat-set ./PE.cfg` makes it fail locally:

In [ ]:
```
!./microbench.exe --stats inst_count.csv --MHz 3500 --reps 25 --function l
```

So run it in the cloud:

In [ ]:
```
!cse142 job run --force './microbench.exe --stats inst_count.csv --MHz 35
```

In [ ]:
```
render_csv("inst_count.csv", columns=columns, average_by="size")  # Compu
```

We are going to see a lot of data like this so, let's be clear about what they mean:

size	IC	CPI	CT	ET	cmdlineMHz	realMHz
size of `array`	dynamic instructions executed	Cycles/instruction	Cycle Time	Execution Time	MHz value from the command line	Measured MHZ

Note that `MHz` and `realMHz` don't quite match. This is due to noise in how we measure elapsed time and "cold start" effects which cause the very first repetition to be much noiser than than the others.

10.3 Meet Your Processor

Let's gather a little bit of information about the CPU we'll be using. We can just ask the OS:

In [43]:
```
!cse142 job run --force lscpu
```

As you can see it's a E-1246G CPU running at 3.5GHz. The model number probably doesn't mean anything to you, but that's what google is for (https://ark.intel.com/content/www/us/en/ark/products/134866/intel-xeon-e-2146g-processor-12m-cache-up-to-4-50-ghz.html).

Question 4 (Correctness - 2pts)

Based on the program output and link above, fill out the table below. Some of the information is in the output above, some you'll need to google for. "Technology node" is roughly the size of the smallest transistors used in designing the chip.:

Parameter	Value
How many physical cores?	[YOUR ANSWER HERE]

Parameter	Value
How many threads?	
Base processor frequency	
Max turbo boost frequency	
Process technology node (nm)	
L1 data cache size (d)	
L1 instruction cache size (i)	
L2 Cache size	
L3 Cache size	

Question 5 (Optional)

Here's some other interesting questions to look into about our processor:

1. What's the Intel code name for our processor's microarchitecture? When was it introduced?
2. What's the maximum clock rate at which processors with this microarchitecture can run?
3. What's the most cores available on a single die with this microarchitecture?
4. What major revision of Intel's microarchitecture is it a part of? How old is this basic design?
5. What do all the things under `Flags` mean?

Here are some resources:

1. https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures (https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures)
2. https://en.wikichip.org/wiki/WikiChip (https://en.wikichip.org/wiki/WikiChip)

10.4 Instruction Count

Let's see how changing the instruction count (`IC`) affects performance. There are two ways we can increase instruction count for `baseline_int()` :

1. We can run the same experiment multiple times.
2. We can increase the size of `array` with the `--size` parameter.

10.4.1 Running The Experiment Mutiple Times

So far we've been passing `--reps` to control how many times we run an experiment. This helps smooth out noise in the measurement, but we can crank it up to increase IC. We'll run it 3 times with 25, 50, and 100 reps.

> **Answer the question *before* you look at the results.** The goal of this question (and many more to follow) is for you to predict the answer and then see if the results match your intuition. Don't be discouraged if you frequently get the prediction wrong: They are intentionally challenging. Also, a major goal of the lab is to highlight behavior that seems *non-intuitive* so you can *improve your intuition*.

Question 6 (Completeness)

In a moment, you'll see four graphs that show how each term of the performance equation changes as we increase `--reps` . What *shape* do you think each curve will have (linear? curved?) and what *direction* will it go (increasing? Decreasing? flat?)? For each term, predict the ratio between it's value at 100 and 25 (i.e., `value_at_100/value_at_25`).

	IC	CPI	CT	ET
Shape	[PUT YOUR ANSWERS IN THE TABLE]			
Direction				
100 vs 25 ratio				

In [ ]:
```
#This takes a while...
!cse142 job run --force './microbench.exe --MHz 3500 --stats 25.csv --reps
!cse142 job run --force './microbench.exe --MHz 3500 --stats 50.csv --reps
!cse142 job run --force './microbench.exe --MHz 3500 --stats 100.csv --re
```

In [ ]:
```
df = IC_avg_and_combine("25.csv", "50.csv", "100.csv")
plotPE(df=df, lines=True, what=[ ('reps', "IC"), ("reps", "CPI"), ("reps"
df[["reps"] +  columns]
```

Question 7 (Correctness - 4pts)

Use the "scratch pad" to compute the actual ratio of the values at `reps = 100` and `reps = 25` and enter them in the table below. How does these results differ from what you expected? (We won't check that you

actually used the scratch pad, but you should. It's a very useful tool.)

	IC	CPI	CT	ET
100 vs 25 reps ratio				
Differences compared to your expectations (if any):				

10.4.2 Increasing the Size of `array`

Instead of increasing `reps` we can increase the size of `array` so we'll run the same computation over more data. We will vary the size over a very large range: 10,000 to 20,480,000.

Kick off the cell below to collect the data. While it's running answer this question:

Question 8 (Completeness)

This time, the four graphs will show each term of the performance equation changes as we increase `--size` . What *shape* do you think each curve will have (linear? curved?) and what *direction* will it go (increasing? Decreasing? flat?)? For each term, predict the ratio between it's value at 5,120,000 and its value at 320,000.

	IC	CPI	CT	ET
Shape				
Direction				
5,120,000 vs 320,000 ratio				

In [ ]:
```
!cse142 job run --force  './microbench.exe --stats array_size.csv --reps
```

In [ ]:
```
plotPE("array_size.csv", lines=True, what=[ ('size', "IC"), ("size", "CPI
render_csv("array_size.csv", columns=columns, average_by="size")
```

Question 9 (Correctness - 4pts)

Compute the actual ratio of the values at 5,120,000 and 320,000. How does these results differ from what you expected? What is the *per-element* (`array`) speedup for a an array of 5,120,000 vs an array of 320,000?

	IC	CPI	CT	ET
5,120,000 vs 320,000 ratio				
Differences compared to your expectations				

Speedup for 5,120,000 vs 320,000-element arrays:

Interesting Question: Why does increasing the size of `array` change CPI? And why does this change occur so quickly?

10.5 Cycle Time

Next, we'll take a look at how clock rate affects performance. Before we do, though, let's see what our options are for clock rate on our machine:

In [ ]:
```
!cse142 job run --force 'cpupower frequency-info -n'
```

As you can see, the processors in our target systems can run between 800MHz and 3501MHz at mostly 200MHz increments.

Let's see how that affects things by plotting execution time as a function of clock speed (we are skipping 3501MHz for the moment. We'll come back to it.). The readings for the current clock speed may vary from run to run. It just ends up at whatever the last experiment left it at.

Kick off the cell below to collect the data. While it's running answer this question:

Question 10 (Completeness)

We are going to plot four graphs that show how each term of the performance equation changes as we increase clock rate. What *shape* do you think each curve will have (linear? curved?) and what *direction* will it go (increasing? Decreasing? flat?)? For each term, predict the ratio between its value at 3500MHz and its value at 1800Mhz.

	IC	CPI	CT	ET
Shape				
Direction				
3500MHz/1800Mhz ratio				

In [ ]:
```
!cse142 job run --force './microbench.exe --stats cycle_time.csv --MHz 80(
```

In [ ]:
```
plotPE("cycle_time.csv", lines=True, what=[ ('cmdlineMHz', "IC"), ("cmdli
render_csv("cycle_time.csv", columns=columns, average_by="cmdlineMHz")
```

Question 11 (Correctness - 4pts)

Compute the actual ratio of the values at 3500MHz and 1800MHZ. How do these results differ from what you expected? How much speedup does double doubling the clock rate provide?

	IC	CPI	CT	ET
3500MHz/1800Mhz ratio				
Differences compared to your expectations:				

How much speedup does doubling the clock rate from 1800MHz to 3500MHz provide? (show your work):

Interesting question: How can clock rate affect `CPI` ?

10.6 Cycles Per Instruction

Unlike `IC` and `CT` we can't set `CPI` directly, but we can adjust the code and see how `CPI` changes. We'll do this in two ways. First, we'll change the data type we are operating on. Then, we'll change the compiler options. Finally, we'll restructure the code.

10.6.1 Floating Point vs Integer Operations

Here's `baseline_double()` (on the left) that is identical to `baseline_int()` (on the right) but uses 64-bit floating point values (of type `double`) point instead of 64-bit integers (`uint64_t`):

In [52]:
```
compare([do_render_code("microbench.cpp", show="baseline_double"),
         do_render_code("microbench.cpp", show="baseline_int")])
```

Out[52]:

```
// microbench.cpp:46-59 (14 lines)
extern "C" uint64_t *baseline_double(uint64_t * _array, unsigned long int size) {
        //double * array = new double[size];
        double * array = (double*)_array;
        for(uint i = 0; i < size; i++) {
                array[i] = 0;
        }

        for (double j = 0; j < 3; j++) {
                for(uint i= 1 ; i < size; i++) {
                        array[i] += i/(1+j)+array[i - 1];
                }
        }
        return (uint64_t*)array;
}
```

```
// microbench.cpp:17-29 (13 lines)
extern "C" uint64_t *__attribute__((noinline)) baseline_int(uint64_t * array, unsigned long int size) {
        //uint64_t * array = new uint64_t[size];
        for(uint i = 0; i < size; i++) {
                array[i] = 0;
        }

        for (uint j = 0; j < 3; j++) {
                for(uint i= 1 ; i < size; i++) {
                        array[i] += i/(1+j)+array[i - 1];
                }
        }
        return array;
}
```

Kick off the the cell below to run both functions, and answer this question:

Question 12 (Completeness)

How do you think each term in the performance equation will change for `baseline_double()` compared to `baseline_int()`?

IC:

CPI:

CT:

ET:

In [ ]:
```
!make  microbench.exe
!cse142 job run --force  './microbench.exe --stats int_double.csv  --reps
```

In [ ]:
```
plotPEBar("int_double.csv", what=[ ('function', "IC"), ("function", "CPI"
          columns=4, average_by="function")
render_csv("int_double.csv", columns=["IC", "CPI", "CT", "ET",], average_
```

Question 13 (Completeness)

How did the results for each term in the PE differ from your predictions (if they did)?

IC:

CPI:

CT:

ET:

Question 14 (Optional)

In `microbench.cpp` there are also `baseline_char()` and `baseline_float()`. Copy the code cells above and modify them to see how those functions compare. What did you find?

Interesting question: How and why do the datatypes we use change `IC` and `CPI` ?

Does anyone else think this version of the "thinking" emoji looks like an alien?

10.6.2 The Compiler's Effect

`microbench.cpp` contains the following function:

In [53]:
```
render_code("microbench.cpp", show="baseline_int_O4")
```

```
// microbench.cpp:31-44 (14 lines)
extern "C" uint64_t *__attribute__ ((optimize(4))) baseline_int_O4 (uint6
4_t * array, unsigned long int size) {
        //uint64_t * array = new uint64_t[size];
        for(uint i = 0; i < size; i++) {
                array[i] = 0;
        }

        for (uint j = 0; j < 3; j++) {
                for(uint i= 1 ; i < size; i++) {
                        array[i] += i/(1+j)+array[i - 1];
                }

        }
        return array;
}
```

It's identical to `baseline_int()` except that for the `__attribute__ ((optimize(4)))` which is a little bit of `gcc` magic to optimize this functions as much as it can (it's the equivalent of passing `-O4` on the command line but just for this function).

Let's see how optimizations affect performance. Kick off the experiment in the cell below and answer this question while it runs:

Question 15 (Completeness)

How do you think each term in the performance equation will change for `baseline_int()` compared to `baseline_int_O4()` ?

IC:

CPI:

CT:

ET:

In [15]:
```
!cse142 job run --force './microbench.exe --stats opt.csv  --reps 25  --MI
```

In [57]:
```
plotPEBar("opt.csv", what=[ ('function', "IC"), ("function", "CPI"), ("fu
render_csv("opt.csv", columns=["function","IC", "CPI", "CT", "ET",], aver
```

Question 16 (Completeness)

Based on the data above, describe in words what affect the optimizations had on the code and the value of each term of the PE.

IC:

CPI:

CT:

ET:

10.6.3 Code Structure

These two functions increment all the elements in an array by `1.0` , but they do it slightly different ways.

In [54]:
```
compare([do_render_code("microbench.cpp", show="matrix_row_major"),
         do_render_code("microbench.cpp", show="matrix_column_major")])
```

Out[54]:

```
// microbench.cpp:123-134 (12 lines)
extern "C" uint64_t *__attribute__((optimize(4))) matrix_row_major(uint64_t * _array, unsigned long int size) {

        double * array = (double*)_array;

        for(uint i= 0; i < size/ROW_SIZE; i++) {
                for (int k = 0; k < ROW_SIZE; k++) {
                        array[i*ROW_SIZE + k] += 1.0; // This Line
                }
        }

        return (uint64_t*)array;
}
```

```
// microbench.cpp:108-119 (12 lines)
extern "C" uint64_t *__attribute__((optimize(4))) matrix_column_major(uint64_t * _array, unsigned long int size) {
#define ROW_SIZE 1024
        double * array = (double*)_array;

        for (int k = 0; k < ROW_SIZE; k++) {
                for(uint i= 0 ; i < size/ROW_SIZE; i++) {
                        array[i*ROW_SIZE + k] += 1.0; // This Line
                }
        }

        return (uint64_t*)array;
}
```

We'll run both versions and compare their performance. Kick off the experiment below and answer this question:

Question 17 (Completeness)

If `size` is equal to 8,388,608 how many times will "This Line" execute in each function? Do you think one will be faster than the other? Why?

How many times does `This Line` execute in `matrix_row_major()`:

How many times does `This Line` execute in `matrix_column_major()`:

Is there any difference in the "Big-O" running time of these two functions?

Do you think one will be faster than the other? Why?

In [11]:
```
!cse142 job run --force './microbench.exe --stats matrix.csv --size 83886
```

In [12]:
```
plotPEBar("matrix.csv", what=[ ('function', "IC"), ("function", "CPI"), (
render_csv("matrix.csv", columns=["IC", "CPI", "CT", "ET"], average_by="f
```

Question 18 (Completeness)

Calculate the speedup of `matrix_row_major` over `matrix_column_major`. Why is this result surprising?

Speedup:

Why is this surprising?:

Interesting Question: Why does the order in which the program performs calculations affect `CPI`?

11 Amdahl's Law

Recall from CSE142 that Amdahl's Law limits the speed up an optimization can provide. It's given as

$$S_{tot} = \frac{1}{\left(\frac{x}{S}\right) + (1 - x)}$$

Where S is the speedup provided by the optimization, x is the fraction of execution time affected by the optimization, and S_{tot} is total speedup.

The function below (also in `microbench.cpp`) calls two of the other functions we have studied.

In [55]:
```
render_code("microbench.cpp", show="everything")
```

```
// microbench.cpp:136-140 (5 lines)
extern "C" uint64_t *__attribute__ ((optimize(0))) everything(uint64_t *
array, unsigned long int size) {
        matrix_column_major(array, size);
        baseline_int(array,size);
        return array;
}
```

In [10]:
```
!make microbench.exe
!cse142 --pdb job run --force  './microbench.exe --stats everything.csv --
```

In [ ]:
```
render_csv("everything.csv", columns=["IC", "CPI", "CT", "ET"], average_by
```

Imagine that you are a manager and your team is tasked with the improving the performance of `everything()`. Members of your team propose two different approaches:

1. Option 1: Replacing `baseline_int()` with `baseline_int_O4()`
2. Option 2: Replacing `matrix_column_major()` with `matrix_row_major()`.

To answer the question below you'll need to look at some csv files you created above. For convenience, you can display them here, like so:

In [17]:
```
display(render_csv("matrix.csv", columns=columns, average_by="function"))
display(render_csv("opt.csv", columns=columns, average_by="function"))
```

Question 19 (Completeness)

Based on the data you've collected, calculate the two speedups below and then decide which of the two options will give the best overall speedup for `everything()` (Show your work).

Speedup of `baseline_int_O4()` vs `baseline_int()`:

Speedup of `matrix_row_major()` vs `matrix_column_major()`:

Which of the two options will give the best speedup?

Question 20 (Correctness - 4pts)

Based on the data you collected earlier in this lab for the performance of `baseline_int()`, `baseline_int_O4()`, `matrix_column_major()`, and `matrix_row_major()`, use Amdahl's law to predict the speedup of each approach. (Show your work, including the values of `x` and `S` and how you computed them.)

Option 1 (Replacing `baseline_int()` with `baseline_int_O4()`) Speedup:

x =

S = (Hint: you computed this in the previous question)

S_tot =

Option 2 (Replacing `matrix_column_major()` with `matrix_row_major()`) Speedup:

x =

S = (Hint: you computed this in the previous question)

S_tot =

`microbench.cpp` has implementaitons of both options:

In [56]:
```
render_code("microbench.cpp", show="option_1")
render_code("microbench.cpp", show="option_2")
```

```
// microbench.cpp:148-152 (5 lines)
extern "C" uint64_t *__attribute__ ((optimize(0))) option_1(uint64_t * ar
ray, unsigned long int size) {
        matrix_column_major(array, size);
        baseline_int_O4(array,size);
        return array;
}
```

```
// microbench.cpp:154-159 (6 lines)
extern "C" uint64_t *__attribute__ ((optimize(0))) option_2(uint64_t * ar
ray, unsigned long int size) {

        matrix_row_major(array, size);
        baseline_int(array,size);
        return array;
}
```

Let's see how they perform:

In [ ]:
```
!cse142 job run --force  './microbench.exe --stats options.csv --size 838
```

In [ ]:
```
render_csv("options.csv", columns=["IC", "CPI", "CT", "ET"], average_by="
```

Question 21 (Correctness - 2pts)

What was the actual speedup for each option? Did Amdahl's law get it right?

Option 1 speedup:

Option 2 speedup:

Did Amdahl's Law get it right?:

12 Power and Energy

Power and energy consumption are both critical design considerations for modern processors, but they are important for different reasons:

1. Power primarily determines the cooling requirements for the system. The more power a system consumes, the hotter it will be and the more cooling it will require.
2. Energy adds to the cost of performing the computation. In practice this is either in energy purchased from a utility (e.g., for a data center) or in battery life (e.g., in your cell phone).

Let's see what can affect the power and energy consumption of a system.

12.1 Clock Rate

Recall the processor power equation from lecture:

```
Power ~ idle_power + C*a*V*f^2
```

and that energy is then

```
Energy ~ ET * (idle_power + C*a*V*f^2)
```

Where `C` is the chip's capacitance (but you think of it as area or chip size), `a` as the activity factor (what fraction of the transistors switch each cycle), `V` is voltage, and `f` is frequency.

We also noted that voltage is generally proportional to frequency, so power consumption is $O(\mathrm{f}^3)$. Hence clock speed has a large impact on energy.

Let's see how the power and energy consumption of one of our functions behaves as we vary the clock rate: `matrix_row_major()`.

The cell below will run `matrix_row_major()` at multiple clock rates between 800MHz and 3501MHz (recall that earlier, we stopped at 3500Mhz. You'll now see why). Kick off the cell below, and while it's running answer this question:

Question 22 (Completeness)

Based on the the power and energy equations what shape of curve (straight? descending? ascending? curved? flat?) would you expect for each of these metrics when plotted against *clock rate*? Where will the minimum be (At small MHz values? large? In between?)?

	shape	Where will the minimum be
Energy vs. MHz		
Power vs. MHz		
Execution Time vs. MHz		

In [ ]:
```
!cse142 job run --force  './microbench.exe --stats cycle_time_energy.csv
```

Run the cell below to plot the data

In [63]:
```
plotPE("cycle_time_energy.csv", lines=True, average_by="cmdlineMHz", what
render_csv("cycle_time_energy.csv", columns=["cmdlineMHz", "realMHz", "ET
```

The first graph shows why we skipped 3501Mhz earlier: The behavior is unexpected. For a `cmdlineMHz` of 3501, we get a measured value of over 3500! This is not a measurement error, it's TurboBoost (https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html). When the system has extra power available and we've requested the highest clock speed possible, the system will give you highest clock speed it can subject to 1) the capabilities of the chip and 2) the cooling capacity of the system.

The rest of the graphs plot `realMHz` on the x-axis, since it actually reflects what the processor is doing.

Question 23 (Correctness - 3pts)

For `matrix_row_major()`, place one 'X' in each row to show which technique will work best to minimize each metric. "Tune clock rate" means that selecting the maximum or minimum clock rate are not good choices.

	Minimize clock rate	Maximize clock rate	Tune clock rate
minimize cooling costs			
maximize battery life			
minimize latency			

12.2 Programs and Energy

We've seen how the code we run and how we compile it can affect both `IC` and `CPI`. But can it affect power and energy consumption? How about clock rate?

To prepare, let's gather some performance numbers about all the functions we've studied (there's nothing new here, it's just all in one place):

In [22]:
```
!cse142 job run --force  './microbench.exe --stats all_functions_pe.csv --
```

In [24]:
```
display(render_csv("all_functions_pe.csv", columns=columns, average_by="fu
plotPEBar("all_functions_pe.csv", what=[("function", "IC"), ("function",
```

Next, we'll measure the power and energy consumption of the same functions.

Kick off the experiment below (which will run all the functions and check their power and energy consumption), and while it's running answer this question:

Question 24 (Completeness)

For each pair of functions, which do you think will consume more power? more energy? Why?

	Power	Energy
baseline_int		
baseline_int_O4		

	Power	Energy
baseline_double_O4		
baseline_double		

	Power	Energy
matrix_row_major		
matrix_column_major		

In [ ]:
```
!cse142 job run --force  './microbench.exe --stats functions_power.csv --l
```

In [44]:
```
plotPEBar("functions_power.csv", what=[("function", "power"),
                                       ("function", "energy"),
                                       ("function", "ET")], average_by="fu
render_csv("functions_power.csv", columns=["ET", "power", "energy"], avera
```

Question 25 (Completeness)

Did the results match your expectations? If not, where? Does the data provide any clues about why?

	What discrepency did you see (if any)?	If yes, why?
baseline_int vs baseline_int_O4		

	What discrepency did you see (if any)?	If yes, why?
baseline_double_O4 vs baseline_double		

	What discrepency did you see (if any)?	If yes, why?
matrix_row_major vs matrix_column_major		

13 Programming Assignment

The labs will all have a programming assignment as part of them. The main purpose of this one is to get you familiar with the autograding submission process. It doesn't require any challenging programming.

In the lab directory, you'll find `hello_world2.cpp` :

In [57]:
```
render_code("hello_world2.cpp", show="main")
```

```
// hello_world2.cpp:3-9 (7 lines)
int main(int argc, char *argv[])
{
        std::ofstream ofs ("hello.txt", std::ofstream::out);
        ofs << "Hello cse142L!\n";
        ofs.close();
        return 0;
}
```

Edit it so that it always writes

```
Hello <your @ucsd.edu email address>!
```

to the file. You can test it with:

In [ ]:
```
!make hello_world2.exe
!./hello_world2.exe
!cat hello.txt
```

And test it in the cloud like so:

In [ ]:
```
!cse142 job run --force ./hello_world2.exe
```

When your submit your code for autograding, it'll run in a more tightly controlled way that let's us reliably measure performance and grade your submission. You can simulate it like this:

In [ ]:
```
!cse142 job run --force --lab intro-bench autograde
```

The results of an `autograde` run end up in the `autograde` directory. Once it's done, you can check the results just like the auto grader dose with:

In [ ]:
```
!./autograde.py --submission autograde --results -
```

If you ran the original code, it gets zero points (`"score": 0`). Once you correctly modify `hello_world2.cpp` , you'll get 1 point.

Once you are happy with your code, commit your changes to `hello_world2.cpp` . You'll have to do this to turn it for official autograding.

In [ ]:
```
!git add hello_world2.cpp
!git commit -m "Yay! I finished the first lab!"
!git push
```

If this asks you for a password, you'll need to interrupt your Jupyter notebook kernel and do it in a shell instead.

14 Recap

This lab has collected real data to understand the how the performance equation, the power equation, and Amdahl's law apply to some simple programs. This exploration presented the following questions:

- Why does increasing the size of array change `CPI` ? And why does this change occur so quickly?
- How can clock rate affect `CPI` ?
- How and why do the datatypes we use change `IC` and `CPI` ?
- Why does the order in which the program performs calculations affect `CPI` ?

Throughout the rest of the course and the labs, we'll find answers to most of these and see how can use those answers to make better use of modern processors.

Question 26 (Completeness)

Which of these questions do you find most interesting and why?

Question 27 (Completeness)

Give three other questions you have after completing this lab.

1. question 1
2. question 2
3. question 3

15 Turning In the Lab

For each lab, there are three different assignments on gradescope:

1. The reading quiz for the lab.
2. The lab notebook.
3. The programming assignment.

In addition, there's a post-lab survey which is embedded below.

15.1 Reading Quiz

The reading quiz is an online assignment on gradescope. It's due before the class when we will assign the lab.

15.2 The Note Book

You need to turn in your lab notebook and your programming assignment separately. There will After you complete the lab, you will turn it in by creating a version of the notebook that only contains your answers and then printing that to a pdf.

Step 1: Save your workbook!!!

In [5]:
```
!for i in 1 2 3 4 5; do echo Save your notebook!; sleep 1; done
```

Step 2: Run this command:

In [2]:
```
!turnin-lab Lab.ipynb
!ls -lh Lab.turnin.ipynb
```

The date in the above file listing should show that you just created `Lab.turnin.ipynb`

Step 3: Click on this link to open it: ./Lab.turnin.ipynb (./Lab.turnin.ipynb)

Step 4: Hide the table of contents by clicking the

Step 5: Select "Print" from *your browser's* "file" menu. Print directly to a PDF.

Step 6: Make sure all your answers are visible and not cut off the side of the page.

Step 7: Turn in that PDF via gradescope.

Print Carefully It's important that you print directly to a PDF. In particular, you should *not* do any of the following:

1. **Do not** select "Print Preview" and then print that. (Remarkably, this is not the same as printing directly, so it's not clear what it is a preview of)
2. **Do not** select `Download as-> PDF via LaTex. It generates nothing useful.

In gradescope, you'll need to show us where all your answers are. Please do this carefully, if we can't find your answer, we can't grade it.

15.3 The Programming Assignment

You'll turn in your programming assignment by providing gradescope with your github repo. It'll run the autograder and return the results.

15.4 Lab Survey

Please fill out this survey when you've finished the lab. You can only submit once. Be sure to press "submit", your answers won't be saved in the notebook.

In [58]:
```
from IPython.display import IFrame
IFrame('https://docs.google.com/forms/d/e/1FAIpQLScHbK7yLlixJqdYsRnpvLLT_
```

Out[58]:

CSE142L Lab Survey

swanson@eng.ucsd.edu Switch account

* Required

Email *

Your email

Student ID (to get credit) *

Your answer

Student ID (to confirm) *

Your answer

Which lab is this? *

Choose

How hard did you think this lab was? *

	1	2	3	4	5	
Very easy	○	○	○	○	○	Ve

How much did you learn from this lab? *

	1	2	3	4	5	
Very little	○	○	○	○	○	A gre

How interesting was this lab? *

	1	2	3	4	5	
Very boring	○	○	○	○	○	Very int

How many hours did you spend on this lab? *

Your answer

Name one thing you liked about the lab *

Your answer

Name another thing you liked about the lab *

Your answer

Name one thing we should change/improve about the lab *

Your answer