Lab 11

Memory Measurement Lab

Due: Week of May 2, at the end of your lab section

This is an individual-effort project. You may discuss the nature of memory technologies and of memory hierarchies with classmates, but you must draw your own conclusions.

In this assignment, you will measure the speed of different memories and draw conclusions based on your understanding of memory hierarchies. **You will be able to complete this lab assignment during lab time.**

The instructions are written assuming you will run the code on your Arduino Nano-based class hardware kit and your account on the *csce.unl.edu* Linux server.

Learning Objectives

After successful completion of this assignment, students will be able to:

- Identify the relative speeds of some types of memory.
- Draw conclusions about a cache's design based on timing data.
- Recognize the performance benefits of cache-friendly code.

Continuing Forward

This final lab assignment should give you a qualitative appreciation of memory concepts discussed in the course.

During Lab Time

You must complete this lab assignment during lab time; it is due at the end of your scheduled lab session. The TAs will be available to answer questions.

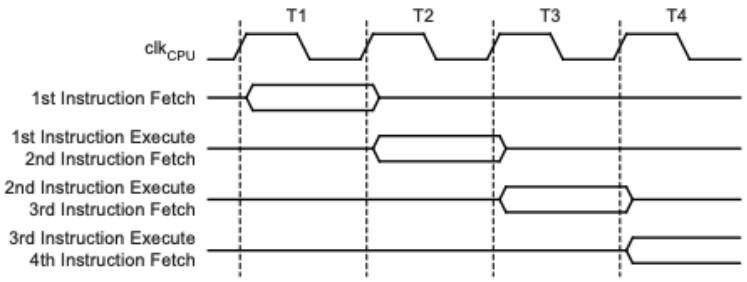


Figure 1: Pipelined instruction fetches and instruction executions. Copied from ATmega328P Data Sheet, Figure 6-4

1 Scenario

Mid-Credits Scene: Herb walks up to you. "Could you do me a favor, please? I'd like to better characterize some of the performance characteristics of the Cow Pi, but I don't have the time to do so. Now that the urgency over developing the security systems is behind us, I need to get back to Eclectic Electronics to clean up a couple of messes that I had to leave there while we developed the Cow Pi. Do you mind measuring the memory performance?"

2 Arduino Memories

The ATmega328P microcontroller on your Arduino Nano has three memories:

- 2KB SRAM data memory
- 1KB EEPROM data memory
- 32KB flash instruction memory

Unlike modern microprocessors, microcontrollers typically don't have cache memories because the CPU and memory speeds are well-matched (for example, the ATmega328P on your Arduino Nano is clocked at 16MHz, about two orders of magnitude slower than the microprocessor in your personal computer).

The instruction memory is designed to deliver instructions quickly, given the predictable access pattern for instruction fetches. This allows the CPU in the ATmega328P to have a 2-stage pipeline, as shown in Figure 1. It is possible to place read-only data in the flash memory;[1,2] however, random accesses to the flash memory will require more than the one clock cycle that instruction fetches enjoy. (Similarly, read-only data can be placed in the EEPROM,[3] but we will not measure the EEPROM's performance in this lab assignment.)

[1] https://www.nongnu.org/avr-libc/user-manual/pgmspace.html
[2] https://www.nongnu.org/avr-libc/user-manual/pgmspace_8h.html
[3] https://www.nongnu.org/avr-libc/user-manual/group__avr__eeprom.html

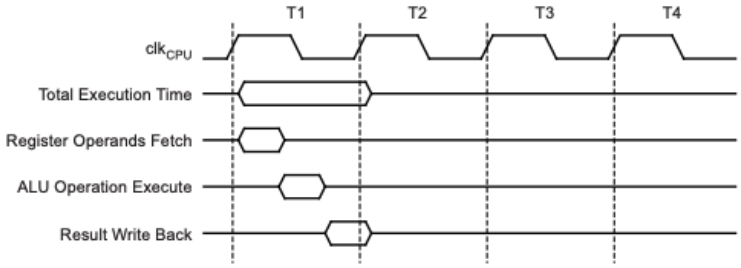


Figure 2: Timing within ATmega328P's "execute" stage. Copied from ATmega328P Data Sheet, Figure 6-5

2.1 MemoryMeasurement.ino

Open *MemoryMeasurement.ino* in the Arduino IDE. The sketch takes four measurements. The **`get_baseline_time()`** function performs every calculation made by the remaining functions *except* for the data retrievals that are unique to each of the remaining functions. The time required for this calculations will be subtracted from the times required for the other functions, allowing us to calculate the time required to retrieve data. The **`time_register_access()`** function retrieves values from the CPU's general-purpose registers. The **`time_sram_access()`** function retrieves values from the microcontroller's SRAM memory. Finally, **`time_flash_access()`** retrieves values from the microcontroller's flash memory.

After you upload the sketch to your Arduino Nano, those four functions will each run once, and the average time to read a value from a CPU register, from SRAM, and from flash memory will be calculated and displayed. If you wish to re-run the sketch, you can press the Arduino Nano's RESET button to restart the program without re-uploading it.

Upload *MemoryMeasurement.ino* to your Arduino Nano now.

Open the *answers.txt* file in a text editor.

2.2 Data in CPU Registers

To draw conclusions about reading data in the CPU's general-purpose registers, two additional pieces of information will be useful. First, calculate the CPU clock's period. Because the clock frequency is 16MHz, you can calculate its period, in microseconds (µs), by dividing $1 \div 16$. Do so and record your answer in *answers.txt*.

Next, note in Figure 2 that retrieving values from the CPU registers is part of the CPU's "execute" stage.

Record in *answers.txt* the amount of time that *MemoryMeasurement.ino* reports is needed to access data in a CPU register. What can you conclude about the time required to access data in a CPU register? Is the reported time significant? Record your answer in *answers.txt*.

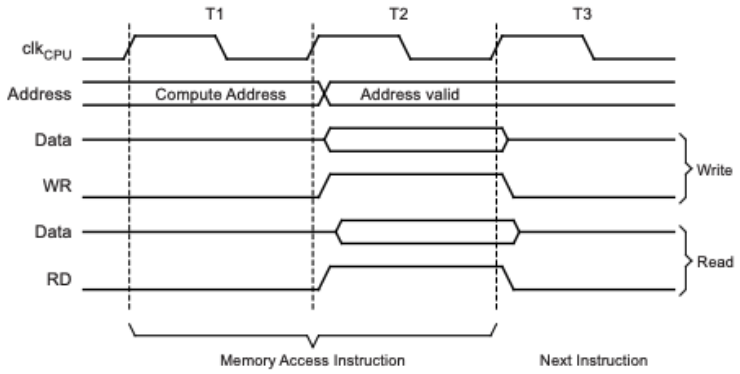


Figure 3: Timing of SRAM Access on the ATmega328P. Copied from ATmega328P Data Sheet, Figure 7-3

2.3 Data in SRAM

Record in *answers.txt* the amount of time that *MemoryMeasurement.ino* reports is needed to access data in a SRAM.

Notice in Figure 3 that the ATmega328P Data Sheet states that accessing data in SRAM requires two clock cycles. Is the reported time to access data in SRAM consistent with the data sheet? Why or why not? Record your answers in *answers.txt*.

2.4 Data in Flash Memory

Record in *answers.txt* the amount of time that *MemoryMeasurement.ino* reports is needed to access data in a flash memory.

The ATmega328P Data Sheet does not describe timing of accesses to flash memory except for instruction fetches, but you can draw conclusions based on the measurements taken.

Approximately how many clock cycles are required to read data from flash memory, according to *MemoryMeasurement.ino*? What conclusions can you draw about the speed of flash memory relative to the speed of SRAM? Record your answers in *answers.txt*.

3 Intel Xeon Caches

We will now take measurements of the cache memories of the microprocessors on *csce.unl.edu*. Upload *CacheMeasurement.c* to your account on *csce.unl.edu*. Compile it with:

```
gcc -O0 -Wall -o CacheMeasurement CacheMeasurement.c
```

You will see two warnings:

```
CacheMeasurement.c: In function ‘measure_helper’:
CacheMeasurement.c:68:13: warning: variable ‘datum’ set but not used [-Wunused-but-set-v
     uint8_t datum;
             ^~~~~
CacheMeasurement.c: In function ‘measure_cache_lines’:
```

```
CacheMeasurement.c:111:22: warning: variable ‘datum’ set but not used [-Wunused-but-set-
     register uint8_t datum;
                      ^~~~~
```

Normally you should correct your code to address warnings; however, in this particular case, we are intentionally updating these variables without using the values they contain.

Open *CacheCharts.xlsx*.

3.1 Measuring Cache Level Sizes

Run the program: `./CacheMeasurement` on *csce.unl.edu*. You will be presented with three options:

```
  1. Measure the sizes of cache levels
  2. Determine the size of a cache line
  3. Exit program
Please choose the measurement you wish to take:
```

For this part of the lab we will use the first option to attempt to discern the sizes of the different cache levels.

Recall that there are three types of cache misses: Cold (or compulsory) misses, in which we can replace an invalid cache block with a valid cache block; Conflict misses, in which a valid cache block must be displaced to allow another cache block to be placed in the cache even though a different cache design might have prevented the conflict; and Capacity misses, in which the working set is too great to fit in the cache, and no cache design could have prevented a conflict. *CacheMeasurement*'s first option will increase the working set size until it is too great to fit in the L1 cache, and then until it is too great to fit in the L2 cache, and finally until it is too great to fit in the L3 cache. The program will measure how much CPU time your process spends waiting for data (it will only measure the time allotted to *your* process, so other process running should have little effect on your measurements). We are less concerned with the specific amount of time, as we are with when there are noticeable increases in the time.

Figure 4 shows a graph similar to the one you will produce. The figure shows the cache size measurements for an Intel Core i7 processor in a 2018 MacBook Pro. This particular processor has a 32KB L1 data cache (also a 32KB L1 instruction cache), a 256KB L2 cache, and a 2MB L3 cache. There are labels showing where these limits are exceeded – other measurement artifacts make it difficult to discern the limit of the L1 cache, but we can use the chart to reasonably estimate the sizes of the L2 and L3 caches with an error less than a factor of 2.

Select the first option, and measure the sizes of the cache levels. Copy the timing data into the "Data" tab in *CacheCharts.xlsx*, in cells C2:C52 (that is, under the heading "first run"). Repeat the first option, copying the timing data under the "second run" heading. Repeat again for a third, fourth, and fifth run. (If you get the error "`Cputime limit exceeded (core dumped)`" then restart the program.) Save *CacheCharts.xlsx*.

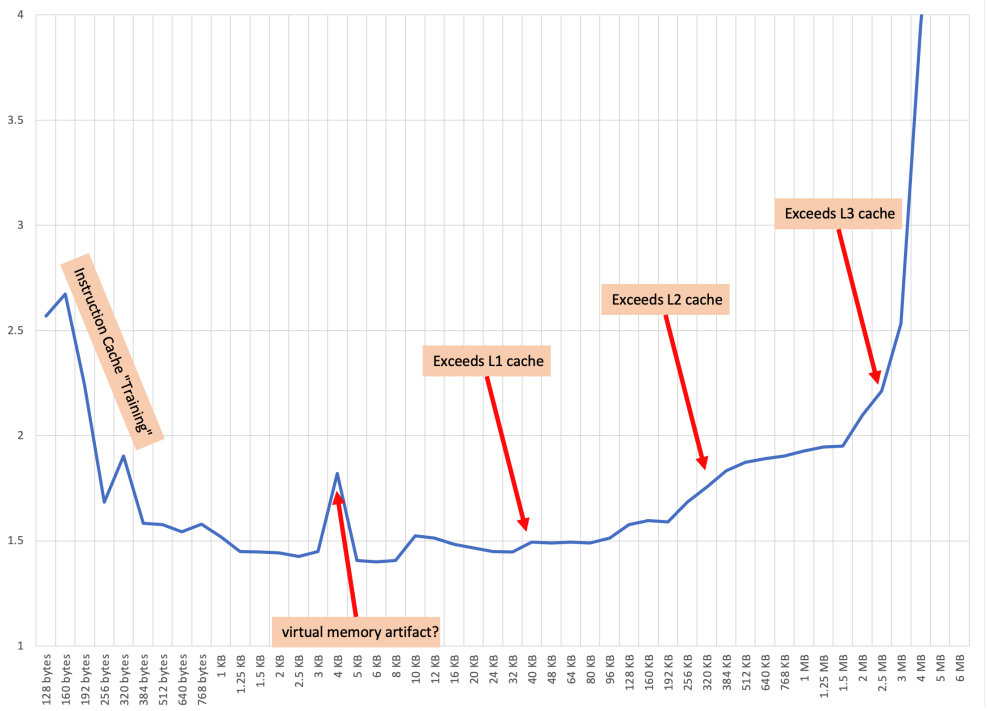


Figure 4: Measurement of cache levels on Intel Core i7 processor.

The "AVERAGE TIME" column throws out the longest and shortest measurements and then averages the remaining three measurements. These averages are plotted in the "Cache Size" tab.

The points at which the working set exceeds the sizes of the L1 and L2 caches should be fairly distinct. Because the L3 cache is shared among 16 cores, other processes (as well as other timing artifacts) will likely result in the L3 cache not having a smooth "plateau" like the i7 processor has in Figure 4.

If there isn't a place on the chart where the timing exceeds the chart's scale, consider editing *CacheMeasurement.c* to use a greater `MAX_WORKING_SET_SIZE`, re-compiling, and re-running the measurements.

Based on the chart, estimate the sizes of the L1, L2, and L3 caches for the Xeon processor in *csce.unl.edu*. Place your estimates in *answers.txt*.

3.2 Measuring Cache Line Size

For this part of the lab we will use the second option to attempt to discern the size of a cache line in the Xeon's cache design.

Recall that caches are designed to take advantage of locality, and when memory access patterns don't take advantage of locality then performance suffers. Your next set of measurements will exploit this fact by increasing the access stride so that you can notice when there is a slight increase in timing due to the stride exceeding the length of a cache line.

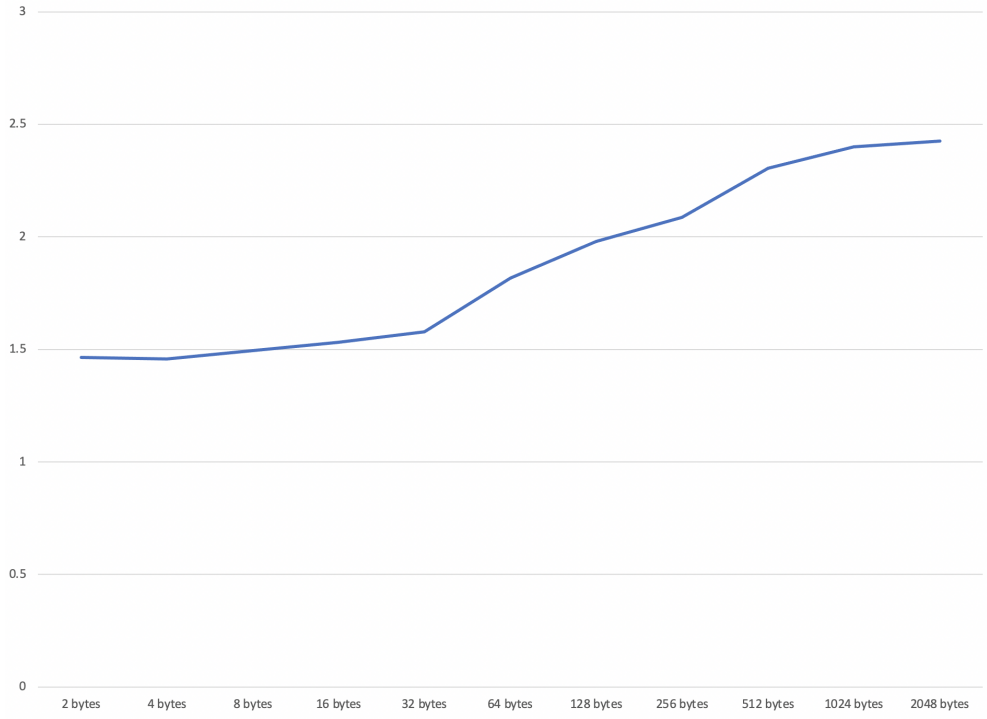


Figure 5: Measurement of cache lines on Intel Core i7 processor.

Select the second option to determine the size of a cache line. Copy the data into *CacheChart.xlsx*'s "Data" tab, cells C64:C74 (that is, under the heading "first run"). Repeat the first option, copying the timing data under the "second run" heading. Repeat again for a third, fourth, and fifth run. Save *CacheCharts.xlsx*.

The "AVERAGE TIME" column throws out the longest and shortest measurements and then averages the remaining three measurements. These averages are plotted in the "Cache Line Size" tab.

If you can discern a point at which the stride exceeds a cache line, then estimate the size of a cache line on the Intel Xeon process, and place your estimate in *answers.txt*. It's very possible that you won't be able to do so. If you cannot, then use Figure 5 to estimate the size of a cache line on the Intel Core i7 processor, and place your estimate in *answers.txt*.

Turn-in and Grading

When you have completed this assignment, upload *answers.txt* and *CacheCharts.xlsx* to Canvas.

This assignment is worth 10 points.

_____ **+1** Drawing a reasonable conclusion about the time required to access data in a CPU register.

_____ **+2** Concluding whether or not the time to access data in SRAM is consistent with the ATmega328P Data Sheet and justifying your conclusion.

_____ **+2** Drawing a reasonable conclusion about the speed of flash memory relative to the speed of SRAM.

_____ **+1** Measuring the cache level sizes and the cache line size on the Intel Xeon processor.

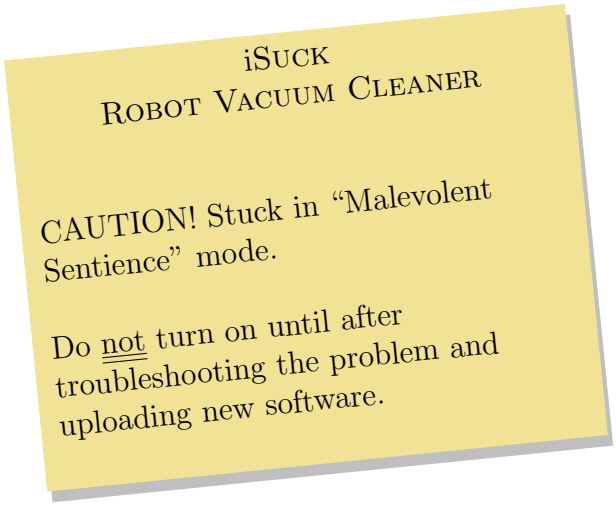
_____ **+2** Drawing reasonable conclusions about the sizes of the Xeon's L1, L2, and L3 caches.

_____ **+2** Drawing a reasonable conclusion about the size of the Xeon's cache line, or about the size of the Intel Core i7's cache line.

Your conclusions do not need to be correct to receive full credit. You can receive full credit if your conclusions are reasonable, given your measurements.

Epilogue

Post-Credits Scene: As an opportunity to stretch your legs, you decided to walk from your office at the Pleistocene Petting Zoo to Eclectic Electronics to personally give Herb your memory measurement data. As you approach Eclectic Electronics' labs, you notice that the lights are flickering, and sparks are dripping from the ceiling. You see that the new Cow Pi-based electronic lock is still holding the lab door shut squarely in its frame – but the door frame has been torn from the wall and is laying askew in the hallway. Peeking into the lab, you see a platform, empty except for a note. The note says:



Cut to black.