# Due: Monday 12/4/15 11:59PM on D2L

## Project Description

Your final project is to optimize the running time of the sieve of Eratosthenes. The sieve of Eratosthenes is a well known algorithm for identifying prime numbers. The algorithm operates by eliminating non-prime numbers from a list of candidates. Effectively, the prime numbers stay in the sieve after the non-prime numbers are sifted out. You can find a detailed illustration of the algorithms on its Wikipedia page: `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`.

Specifically, your objective is to optimize `sieve.asm` – a MIPS implementation of the algorithm. To reduce the running time of your code, you may apply techniques such as reordering instructions, reducing the power of instructions, optimizing data layout in the cache, and even optimizing the algorithm itself. The techniques you use are up to you, there are however three things you are not allowed to do. First, you are not allowed to pre-compute the primes and simply have your code print them out. In other words, your algorithm has to do actual computations of the prime numbers. Second, you are not allowed to submit MIPS code that has been generated by a compiler. In other words, all the assembly code you submit has to be generated by hand. Finally, third, you are not allowed to submit code that you have not written yourself, or with your partner – we will check your submissions for originality. However, you may look online for code optimization techniques other then the ones we went over in class and adapt code examples from web pages, tutorials, or news group posts.

To lower the learning curve and the frustration of working with full-feature hardware simulators, such as Qemu or Simics, you will optimize your code for the MARS simulator you have used in your homework assignments. One downside of working with MARS is that it does not simulate the five stage pipeline, and so it does not report the number of clock cycles your program takes to execute. You will estimate the running time of your program in cycles based on the number and type of instructions executed, and based on the cache performance of your program.

To calculate the number of cycles required to execute the instructions in your program you will use the MARS Instruction Statistics tool and the Data Cache Simulator tool. Both these tools can be accessed from the MARS tools menu (Figure 1). Figure 2 shows the number of instructions required for the unoptimized version of `sieve.asm`. To sum up the number of cycles required by the executed instructions you will assume a pipelined implementation with forwarding (consult figure Figure 4.51 in your text), in which each ALU and Jump operation take one cycle. Assume each Branch operation, whether taken or not, take two cycles. Further, assume that memory operations take two cycles on cache hit, and forty cycles on a cache miss. To determine the number of memory instructions that result in cache hits and misses you will consult the Data Cache Simulator Tool, shown in Figure 3. You will use the default settings of a directly mapped 128 B LRU cache with 8 blocks of 4-word each. Finally, assume that the the multiplication instructions (and division instructions you choose to add any) take five cycles each. These instruction are reported in the Other category of the Instruction Statistics tool. You will also observe a discrepancy between the Memory Access Count of the Data Cache tool and Memory instruction count of the Instruction statistics tool. The Data Cache tool correctly classifies the `la` instructions as data instructions, while the Instruction Statistics tool groups these instruction in the Other category. So, to correctly compute the number of multiplication instructions, you will need to subtract out the `la` operations.

To make computations of program clock cycles easier I have created a spreadsheet `running_time.xslx` you can use to evaluate your programs. Figure 4 shows the calculation of the clock cycles taken by the unoptimized `sieve.asm` to find all the primes less than 200. Before copying the numbers from the output of the MARS tools remember to Reset the counts before every execution of the program.

## Grading

Your project grade, out of 100 points, will depend on the running time (total number of cycles) of your optimized `sieve.asm`. You will receive 100 points if you reduce the running time of the unoptimized implementation by a factor of 6. 90 points for a factor of 5 reduction. 80 points for a factor of 4 reduction. 70 points for a factor of 2 reduction. If you are not able to reduce the running time by at least a factor of two, your grade will depend on the factor of improvement you are able to achieve. The team with the lowest running time will receive a bonus of 5 points added to their score.

## Submission Instructions

Submit your work into the Dropbox on D2L into folder "Final Project."

1. Find a **partner** and submit their name as `your_name_partner.txt`. Please contact me if working with a partner represents a significant hardship for you, otherwise I expect you to work in pairs.
2. Every Friday from now until the end of the course submit a partner evaluation by the end of the lecture. Each partner will record the percentage of the work/effort/time they are putting into the project. If both partners feel they are doing 60% of the work, please record that, though ideally the percentage should add up to a hundred. Both partners will sign the sheet.
3. Submit a list and short descriptions of the optimizations you have used in your program as `your_name_optimizations.txt`.
4. Submit your final program as `your_name_sieve.asm`. Both partners will submit the same solution file(s). Please beware that programs which do not assemble in MARS will loose 50% of credit before partial credit is assessed.

   Remember that the late policy is 10% of total points deducted for every day the submission is late.

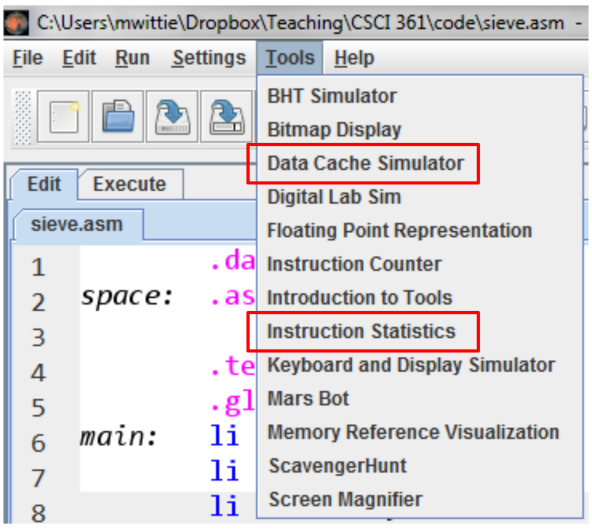


Figure 1: Tools menu.

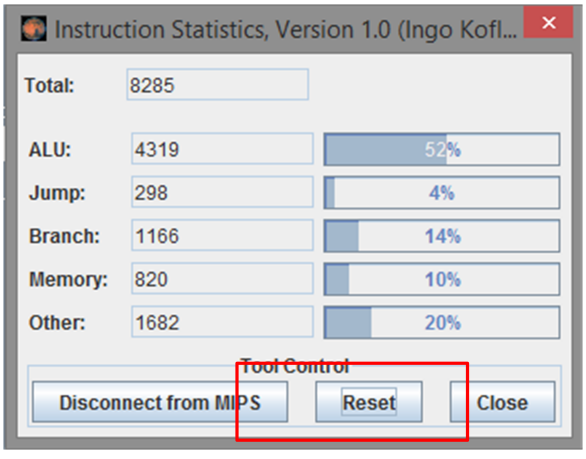


Figure 2: Instruction statistics after program execution.

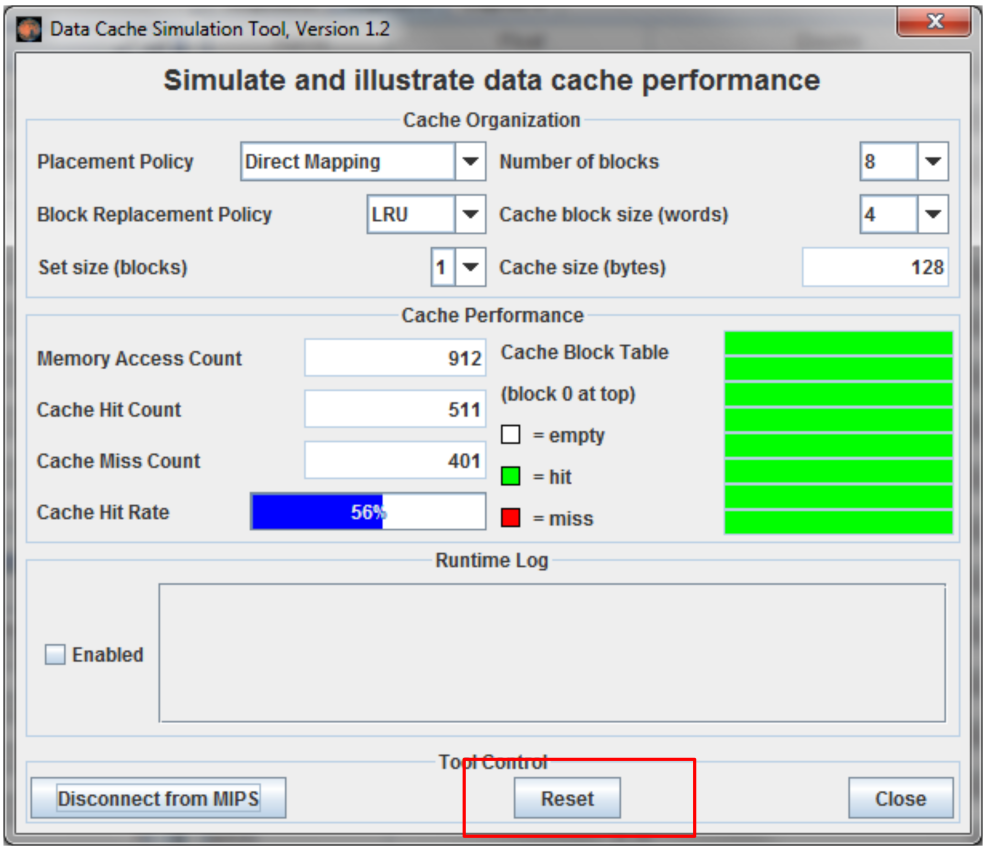


Figure 3: Data cache statistics after program execution.

| **MARS Tool Output** | | | **Calulations** | | |
|---|---|---|---|---|---|
| | | | | | |
| **Instruction Statistics Tool** | | | | | |
| Instruction type | Count | | Adjusted count | CPI | Total cycles |
| ALU | 4319 | | 4319 | 1 | 4319 |
| Jump | 298 | | 298 | 1 | 298 |
| Branch | 1166 | | 1166 | 2 | 2332 |
| Memory | 820 | | | | |
| Other | 1682 | | 1590 | 5 | 7950 |
| | | | | | |
| **Data Cache Simulation Tool** | | | | | |
| Access | Count | | | | |
| Cache hit | 511 | | 511 | 2 | 1022 |
| Cache miss | 401 | | 401 | 40 | 16040 |
| | | | | | **31961** |

Figure 4: Calculation of program running time using `running_time.xslx`.