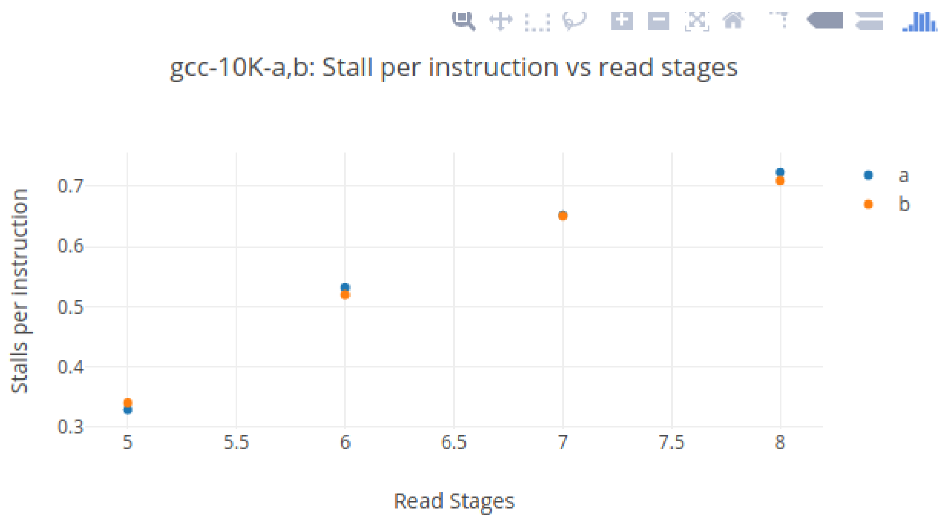
Lab 1 Report, Hazard Detection
Zachary Bechhoefer, Jake Drake, Benjamin Spenciner
03/29/2018

Hazard detection is the process by which conflicts in resource access are discovered. These resources are registers holding data in a program, being read and written to for operations. Because registers are typically involved in multiple operations running both in parallel and sequentially, it is possible for a resource to be requested prematurely. Resource usage violating the order of operations detailed by assembly will result in stalling, which slows down programs. These hazards can be handled by the programmer or compiler via hazard detection analysis, followed by rescheduling operations to avoid stalling altogether.  The purpose of this lab was to explore the outcomes of, and the challenges to hazard detection in assembly execution.

To control the scope of the experiments, we only test for WaR hazards (data dependencies) in this lab. A test was run on a series of trace files containing a record of assembly instructions. The test iterated over the instructions in the trace, and checks for instances of data dependency. Our modulated variables are the stages to the MIPS pipeline, and read stages. The read stages is the primary factor in data dependency, because it determines when a register read is complete, and the register is available again. While MIPS only modifies data dependency simulation cosmetically, read stages determines how many clock cycles , and therefore instructions can pass before computation can continue.

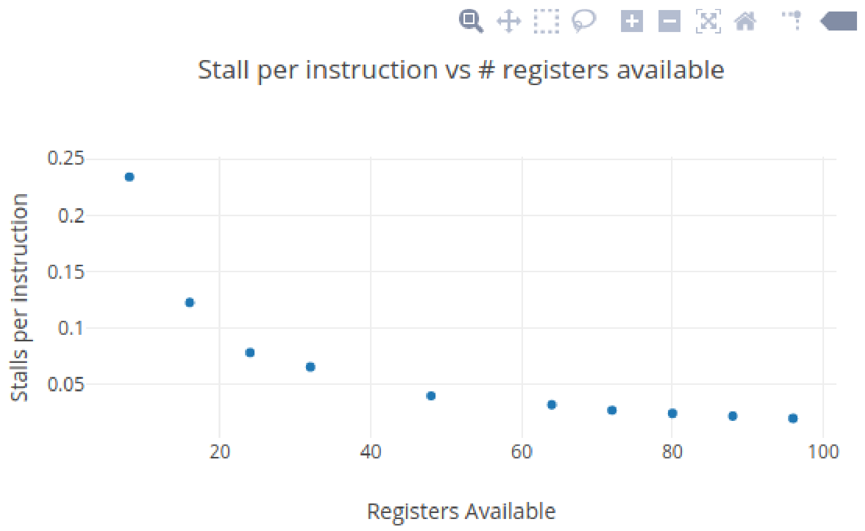
In our test software, we simulate assembly computation (without executing code) in terms of dependency alone. This is achieved by tracking clock cycle, recording stalls on instructions, and relating instructions to previous instructions inside the number of read stages. Because the only hazard we are investigating is WaR, we can simply relate the write register of one instruction to the read registers of previous instructions. This is simple in theory, but in our programming language (C++), the test software required some finessing with relatively low-level data structures. However, this may add considerable realism, since compilers that optimize their assembly and hazard detection are usually also written in C/C++.

An experiment we ran was measuring Stall cycles as a function of MIPS pipeline length, and read stages. The results from two pseudo-random traces is as follows:



The results of this experiment demonstrated no significant change in stalls per instruction (SPI) from changes in MIPS length. This is sensible, since with MIPS lengths of n and n+1, the the ideal pipeline of ten thousand instructions changes by (1/(10090+n))-(1/(10000+n+1)), negligible. Therefore, it is apparent that MIPS pipeline length has a negligible effect on SPI. On the other hand, changing read stage length appears to increase SPI. While the rate of SPI increase as a function of read stages decreases, it does not stand to reason that it is convergent. In fact, with an infinitely long read stage, it can be expected that you have an infinite SPI. This is a worthwhile note for IC engineers, who can know to sacrifice MIPS length to save on read stages, when/if possible.

Another experiment measured SPI versus the number of registers available, as follows.



These results are expected. A small number of registers force all the computation into a smaller register-space, causing a computing equivalent of crowding. Likewise, having an arbitrarily large number of registers sees diminishing returns for computations of finite complexity. Conceivably, there could be computations that utilize all 96 registers in the final test case. There is indeed a small improvement in SPI for those last few registers, but the benefits are minimal. Clearly, the number of registers in an IC should reflect expectations for the complexity of computation it will perform.

The output of our test program speaks to the importance of hazard detection. Searching for data dependencies alone, hazards could add 50% more (read stages of just two) cycles to program execution. Given that there are other kinds of dependencies, the potential total loss could easily double execution time.  It is apparent that while hazard detection for unfamiliar C programmers may be inconvenient. That being said, the cost of running hazard detection and operation rescheduling, appears to be  minuscule compared to time saved from hazard avoidance across many executions of a program.