Computer Architecture and Mobile Processor Programming assignment #4:
pipelined MIPS emulator with cache

Introduction

Now, we will build a pipelined MIPS emulator with the cache. With respect to proper technical scaling, we need to improve bottleneck performance, that is memory latency/ bandwidth. In Von Neumann computers, memory access is very frequent because all instruction and data are stored in memory, and memory access latency is usually hundred times slower than register access latency. The latency to memory access introduces the stall cycles in the pipeline. To mitigate the slow memory access latency, modern CPUs use a cache storage, which is small, but fast storage inside CPU.

For a processor with cache, memory access is firstly checked with the cache, before the memory is accessed. If data resides in the cache (cache hit), data is directly accessed from the cache, instead of memory. If the data is not in the cache (cache miss), CPU fetches data from memory. Note that cache is within the CPU so that it can be quickly accessed from the pipeline.

The cache leverages the locality of memory access pattern. That is, some memory address is more probable to be accessed than the others. Thus, placing such memory region in the closer and faster location such as cache can effectively improve performance. The locality can be understood in two different perspectives: spatial locality and temporal locality. Spatial locality is the locality of the observed in accessed memory addresses. That is, the nearby memory address is more probable to be accessed than farther addresses. Temporal locality is the locality of the observed in accessing times of memory region. That is, recently accessed memory is more probable to be accessed again than the other memory regions.

To best utilize cache storage, we need to define several policies for it. First, we need to define a cache structure, including cache line, set-associativity, and the number of entries in the cache. To maximize spatial locality, cache block (or cache line) contains some nearby bytes from the requested location. All data in the same cache line is fetched from memory and stored back to memory at the same time. The entire cache consists of multiple cache lines.

Because cache storage is much smaller than the entire memory size, we need additional information that remembers the memory address, where the current cache line came from. The additional information is called as a tag of a cache line. That is, the tag of a cache line represents some bits of the memory address. There are several mapping schemes of memory region onto cache line: direct-mapping, fully-associative mapping, and set-associative mapping. To reduce the size of tag bits, the cache line index can be further considered. Cache has multiple cache lines, and the index of the cache line can be determined by some of the address bits. For direct mapping, the cache line index is fixed, and the cache line entry is determined according to the address. For fully-associative mapping, the cache line index is irrelevant to the address. That is, the memory can go anywhere in the cache line, regardless of the address. Set-associative mapping is a hybrid cache-memory mapping; some cache lines are grouped as a set, and the set index is calculated similarly with direct mapping.

Second, replacement policy has to be determined. Because the cache is smaller (and expensive) than memory, it cannot store entire data in the memory. Instead, it can only store a partial range of memory. When the cache is fully utilized, and another cache miss occurs, cache conflict

miss occurs. Because already all cache line entries are utilized, there is no additional room for storing data. In the case, one of the entries (cache line) has to be evicted from the cache, and the requested memory region has to be fetched into a cache line. Thus, the cache has to select which cache line has to be replaced. A good rationale for replacement entry is maximizing temporal locality. LRU (Least Recently Used) algorithm is used to select the least recently accessed entry in the cache. That is, LRU tries to remain the most probable entries that could be accessed again in the cache.

Finally, write policy has to be determined. When a cache line is updated, the write operation can be applied either on the cache line (write-back) or through the memory (Write-through). When we write data only on the cache line, the memory update is delayed, and memory- cache consistency is broken. (That is okay for single processor system) When we write data to the memory through the cache, memory access speed is reflected to pipeline execution.

Computer Architecture and Mobile Processor Programming assignment #4:
pipelined MIPS emulator with cache

In this programming assignment, you will simulate cache operations along with your pipelined MIPS, fully in software. You have to implement above-mentioned cache structure with the proper assumptions.

Since this program assignment could be one of your major take-outs from this course, so please work hard to complete the job/semester. If you need help, please ask for the help. (I am here for that specific purpose.) I, of course, welcome for any questions on the subject. Note for the one strict rule: Do not copy the code from any others. Deep discussion on the subject is okay (and encouraged), but the same code (or semantics) will result in the sad ending.

Extra implementation/analysis is highly encouraged. Unique /creative approaches are more appreciated, even trial can get credits. For example, you can implement various LRU algorithms, or you can conduct cache performance analysis with respect to cache size, set-associativity. The multi-level cache can also be considered. If your pipeline implementation is not stable, you can implement cache with single cycle project.

Last, but not least advice: Begin as early as possible. Ask for help as early as possible. Try as early as possible. Those are for your happy-ending.

The followings are specific requirements for your program.

1. The objective: understand cache structure and analysis on cache performance.
   A. Your program should produce the correct output.
   B. Compare the cache hit/miss and average memory access time (AMAT).
2. Machine Initialization:
   A. Before the execution, the binary file is loaded into the memory. Note that memory can be a data structure defined with a large array. Read all the file content into your memory (data structure). Assume that all register values are all zero, except for RA, of which value is 0xFFFF:FFFF. Thus, when your PC becomes 0xFFFF:FFFF, your machine completes execution and halts. Your application is loaded to 0x0, and the stack pointer is 0x100000.
   B. The cache also has to be clean; all entry is initialized as zero.
3. Example assumptions:
   A. Cache line size is 64 bytes. Set-associativity can be various (Direct-mapped, 2-way, 4-way, 8-way, etc.), configured before the execution. Cache size (data store size) can also be configured before execution (64 bytes, 128 bytes, 256 bytes).
   B. The cache should implement proper replacement algorithm.
   C. The cache should implement proper write policy.
   D. Cache access latency is one CPU clock cycle time; memory access latency is 1000 cpu clock cycle time.
4. Output: At the end of program execution, the simulator prints out the statistics from the execution. Statistics may include
   A. total number of cycles of execution
   B. number of memory (load/store) operations
   C. number of register operations
   D. number of branches (total/taken)

due by June 15th

   E. cache hit/miss (cold miss or conflict miss)
5. Program completion/terminal condition:
   A. At the end of the execution, you need to print out the calculated result value. We know the end of execution by moving PC to 0xFFFF:FFFF. Along with the calculated result, you should print out the execution cycles of the emulator.
   B. The final return value is stored in v0 (or r2) register.
6. Evaluation:
   C. Different credits are given for implementations and demos.
7. Input program binary: You can use your own MIPS code (compiled) or download binary.
   D. To make MIPS binary, use MIPS cross-compiler toolchain. First, write C code, and compile mips-Linux-gnu-gcc with –c option (compile only). Second, translate the object binary, stripping ELF headers. mips-Linux-gnu-objcopy –O binary – j .text input.o input.bin. Third, check the integrity between binary files: objectdump, mips-linux-gnu-objdump –d input.o , vi –b input.bin ¼ check with :%!xxd option. (google internet) or hexedit tool
   E. To use with multiple functions, you need to hand-carve binary for function calls (jal 0, originally). For example, jal to 0x40 can be encoded as (0x0C000010).
   F. Sample input file examples can be downloaded from the e-learning site. Two representative example programs are 1) summation from 1 to 10, 2) calculating 4 factorial.
   G. Some good example can be sorting large data arrays.

Enjoy MIPS emulator programming!

Good Luck!

due by June 15th

Computer Architecture and Mobile Processor Programming assignment #4:
pipelined MIPS emulator with cache

due by June 15th