What are you expected to deliver for this assignment?

First, perform a *git pull* on your cse4302 cloned GitHub repository. A new directory, named lab3 will be added. It has the necessary files to complete this programming assignment. You will be modifying simulator.c to complete the multi-cycle 5-stage pipelined implementation for the riscy-uconn ISA. The details on the ISA and what modifications you need to make to the simulator.c are outlined in the next sections of this document. Your implementation must be fully functional for all testcases provided in the unittests directory. You are expected to compile each testcase using the assembler provided to you in assembler directory. For example, for lw_sw_test1.asm testcase, you will generate a lw_sw_test1.out by running it through the assembler. Then, you may run this testcase using the simulator command: ./simulator lw_sw_test1.out 0 0 when you don’t want forwarding logic and cache enabled, ./simulator lw_sw_test1.out 1 0 when the forwarding logic is enabled but cache is disabled, ./simulator lw_sw_test1.out 0 1 when the forwarding logic is disabled and cache is enabled, and ./simulator lw_sw_test1.out 1 1 when both forwarding logic and cache are enabled.

Once you are ready to submit your simulator.c file for this programming assignment, you are required to email it to the instructor at khan@uconn.edu before the deadline. You will receive an acknowledgement email back from the instructor if your submission is accepted. It is your responsibility to ensure a timely submission of your programming assignment. In the subject line, you must state PA3 submission: <yourfullname>. You will need to schedule a 10-15 minute time slot with the instructor to complete a demo of your assignment to get credit. This process must complete by the last day of classes.

2 Multi-cycle 5-Stage Pipeline Simulator

The objective of this programming assignment is to extend the 5-stage pipelined implementation from Lab2 to support multi-cycle data memory operations. The assignment involves two features that you will implement: (1) extend the five pipeline stages to handle each LW and SW instruction execute its data access operation by stalling in memory stage for 10 cycles; (2) implement a direct-mapped single-cycle cache to improve the data access latency of LW and SW instructions in the memory stage. The details of these features are described assuming that you have a functionally correct implementation of Lab2.

The main() function is extended to report cdump() that prints the status of cache at the end of simulation. Another print statement reports the data memory stats using the newly added dcache accesses and dcache hits global variables. It is your responsibility to precisely track these counters as described in subsequent subsections. The main function also handles the processing of forwarding enabled and cache enabled command line arguments.

In this lab, you will modify the following functions in simulator.c: process_instructions(), decode(unsigned int instruction_fetch), memory_stage(struct Registers alu_out). In addition, two new functions, dcache_lookup(int addr_mem) and dcache_update(int addr_mem) are added for Lab3 that you need to complete. Your submission is required to not modify any of the other functions in simulator.c. Moreover, you are not allowed to modify the contents of simulator.h, mipsInstructionMap.h, and the Makefile.

2.1 Stalling memory_stage

The memory_stage() function needs to be extended from your Lab2. For both LW and SW instructions, this stage handles the memory access latency, which is now determined by the variable cycles_to_access_dmem specified in simulator.c. Since this variable is set to 10 cycles, the memory_stage() must stall for ten cycles before allowing the LW or SW instruction to progress to the writeback stage(). Two new global variables (dmem_busy, dmem_cycles) to manage memory_stage stalling are added in simulator.h. dmem_busy must be de-asserted at the start of each memory_stage() function evaluation. This is done to ensure this flag is only asserted explicitly for cycles where it will evaluate true, as described below. The dmem_cycles counter is used to track the number of actual stall cycles in the memory_stage(). When it is less than cycles_to_access_dmem, the dmem_busy must be asserted, the current instruction memory_stage() converted to a NOOP, and memory_stage() function must be returned. Asserting dmem_busy ensures that the pipeline stalls whenever the memory_stage() is stalled waiting for the data access to complete. When dmem_cycles is
equal to cycles_to_access_dmem, the dmem_cycles counter is reset, and then the memory access as described in Lab2 is carried out before sending the LW or SW instruction to the write-back stage.

The memory_stage() function also needs to track the dcache_accesses and dcache_hits counters for statistics purposes. Since this phase of your Lab3 has not introduced the cache, all LW and SW instructions must update the dcache_accesses counter when dmem_cycles is equal to cycles_to_access_dmem. This ensures that the dcache_accesses counter tracks a precise count of all LW and SW instructions being processed successfully in the simulator. The dcache_hits should report zero for this phase of Lab3.

When dmem_busy is asserted, the process_instructions() must update the pc, instruction_fetch, decode_out, alu_out and mem_out state so that the pipeline holds the state of fetch, decode, execute and memory stages. This is done to ensure the pipeline stages are able to re-execute their current instructions during memory_stage stall cycles.

The decode() function needs to be extended from your Lab2. When dmem_busy is asserted, the decode() function must assert pipe_stall before injecting NOOP in decode() output. This will ensure that decode and fetch stages stall. Moreover, when dmem_busy is asserted, the decode() function must return before any modifications are made to the pc or br_taken variables. This will ensure that the program counter does not update, and the branch logic does not introduce any side effects as the pipeline stalls the fetch and decode stages.

The execute() and write_back_stage() functions do not need any modifications beyond Lab2’s code. Once the above modifications are in place, the pipeline implementation is ready to execute the unitests under the cache-enabled = 0 command line option.

2.2 Improving LW and SW access latency using a Cache

The cache model for this assignment is fixed using the following properties: (1) the cache is a direct-mapped cache that uses the 32-bit address to lookup the word-addressable memory. (2) the cache line size is fixed to 64 bytes or 16 words. (3) the number of cache lines is fixed to 16 using #define NUM_LINES in simulator.h. (4) the cache hit latency is fixed to 1 cycle, while cache miss incurs cycles_to_access_dmem (= 10) number of cycles. Using (2) and (3), the cache size is set to 1024 bytes.

The cache model consists of the tag, while the data is retrieved using the memory. Each cache line in the tag is modeled using the cblock struct in simulator.h, which contains the valid and tag state information for each cache line. The cache model is initialized in the initialize() function for you (see lines 368–378 in simulator.c). The cache initially consists of 16 cache lines, and for each cache line the valid and tag information is initialized to zero. This indicates that at startup the cache is empty. Two functions, dcache_lookup(int addr_mem) and dcache_update(int addr_mem) need to be completed to implement the cache lookup and update functionality. Note that memory is word-addressable, hence the minimum number of bits extracted from memory are 32-bits of data. On a cache lookup or update, the cache model needs to address one of the 16 cache lines. Note, each cache line size is 64 bytes, or 16 words. You must decode the appropriate bits from the address to track the offset and index for the cache. The remaining most significant address bits need to be tracked as tag bits.

The cache_update() function takes the memory address, and uses the index bits to set the associated valid bit to 1, and capture the tag bits in that cache location. The cache_lookup() function takes the memory address, and lookup the cache using the index bits to extract the tag in that cache location. Then based on the tag from the address, and the valid and tag information extracted from the cache, this function must resolve the local variable “hit”. In case of a cache hit, this function will return a 1, otherwise the cache missed and it returns a 0. You need to implement the above functionality by updating the code for functions dcache_lookup(int addr_mem) and dcache_update(int addr_mem).

The cache model is used in the memory_stage() function. For a LW or SW instruction, the memory_stage() must perform the cache_lookup() based on the effective address and determine if this instruction hits or misses in the cache. If the cache_enabled is set, and cache_lookup() returns a hit, the memory_stage() does not stall for the LW or SW instruction and continues to perform memory access and proceed to write-back stage. In this scenario, the dcache_hits counter must be incremented to precisely track this LW or SW instruction as a cache hit. In case the cache_lookup() returns a miss, the memory_stage() handles the memory lookup by stalling the pipeline for cycles_to_access_dmem (= 10) number of cycles. This scenario must be handled in a manner described in the previous subsection. However, when the dmem_cycles equals cycles_to_access_dmem, the dcache_update() function must be invoked before accessing memory. This is done to make sure the cache tracks this address for future cache hits.

Once the above modifications are in place, the pipeline implementation is ready to execute the unitests under the cache_enabled = 1 command line option. It is your responsibility to make sure appropriate handling of the cache_enabled flag is implemented in the memory_stage() function.