In Assignments 3 and 4, you will build a dynamically-scheduled multiple-issue processor. This assignment starts you off towards this goal by building the basic components in the context of a single-issue, in-order processor with multiple functional units. These components will be extended with dynamic scheduling and multiple issue in Assignment 4.

Background

For this exercise, we will use a different processor core architecture than we have been using in Assignments 1 and 2. The architecture is similar to that introduced in class for implementing Tomasulo-like algorithms for dynamic scheduling.

The processor core consists of instruction scheduling logic interacting with multiple functional units (FUs) and the register set using a register update unit (RUU), which acts as a combination reservation station store and reorder buffer) and a load/store queue (LSQ). The instruction scheduling logic consists of the following stages:

- **Instruction Fetch**: This stage fetches instructions and puts them into a fetch/dispatch queue. For this assignment, we shall assume that we can fetch one instruction/cycle.

- **Dispatch**: This stage retrieves an instruction from the fetch/dispatch queue, allocates an entry in the RUU (and the LSQ if required), checks for data hazards and if none exist, then inserts the instruction into a ready queue.

  In this assignment, we require that all operands of an instruction be ready before it is placed in the ready queue. This implies both that output registers are available for writing, and that input registers are available for reading. The processor keeps track of registers that are destinations for instructions under execution using a create vector structure, which identifies the RUU entry that will be writing into it.

  Special handling is required for load/store instructions as discussed in additional detail below.

- **Issue**: This stage examines the instruction at the head of the ready queue, and if functional unit resources are available, then issues the instruction to the appropriate FU. Each FU is modeled in terms of two parameters: an issue latency and an operation latency. The former determines how frequently instructions can be sent to the FU, while the latter indicates when operation results will become available.

  Special handling is required for store instructions as discussed below.

- **Writeback**: This stage waits for an instruction to finish execution and updates the register set with the result. Instructions in the dispatch stage, waiting for this result can be made ready at the end of the same cycle.

- **Commit**: This stage is responsible for “retiring” instructions at the head of the RUU by committing the results of the instruction to architected processor state. The RUU (and optionally the LSQ) entry is freed up.

  Special handling is required for store instructions as discussed below.

In this assignment, you will simulate the behavior of the above processor core using the SimpleScalar toolkit. The dispatch stage of the core is responsible for functional simulation of instructions; the rest of the stages just simulate the instruction flow through different parts of the core.

Assignment 4 works with the same basic structure as above, but allows the fetching, dispatching, issue, writeback, and committing of multiple instructions every cycle. Furthermore, the instruction dispatch logic is altered to implement dynamic scheduling.
For simplicity, we shall assume the existence of a
- Perfect instruction cache (no I-cache misses)
- Perfect data cache (no D-cache misses)
- Perfect branch predictor, implemented as discussed below.

**The Assignment**

The assignment requires you to provide simulator code for implementing the five instruction scheduling stages described above, so as to realize a single-issue, in-order processor with multiple functional units. Since the functional units have different issue and operation latencies, instructions may end up producing results out-of-order (even when issued to the FUs in order); however, the commit stage ensures that instructions finish execution in order.

To help you get started, I have provided a sketch of the simulator, `sim-multfu.c`, which defines the various structures, their sizes, and functional unit configurations. You will need to update the makefile rules as in the previous assignments to make this file part of the `simple-sim-3.0` sources (follow the pattern of rules for the `sim-outoforder` simulator).

**Implementation notes:**

1. **Implementing perfect branch prediction:** As in Assignments 1 and 2, the instruction fetch stage works with a variable, `fetch_pc`, which represents its knowledge of the next address from which instructions need to be fetched. Perfect branch prediction can be implemented simply by updating this variable with the correct PC value obtained by functionally simulating the instruction. Since the execution of the dispatch stage precedes that of fetch, this update has the effect of ensuring that no slots are lost because of mispredictions.

2. **Interacting with functional units:** To help emulate functional units and the fact that they can be configured with different issue and operational latencies, we rely on a SimpleScalar module, `resource`, which models such units. The API for this module provides functions to create a pool of FUs, to allocate a FU as required, and to keep track of when this FU will next become available for issue.

   The supplied skeleton code in `sim-multfu.c` includes code fragments showing how to create, initialize, and interact with the resource module. Additional details can be found in the `sim-outoforder` sources.

   To model execution latency in a FU, we rely on an event queue structure. This structure allows the enqueuing of “result” events, which become ready a specified number of cycles in the future. The simulator uses this functionality to indicate when an instruction should be deemed to enter its writeback stage.

3. **Dealing with load/store instructions in the processor core:** Memory instructions are handled somewhat differently as described below.

   A load or a store instruction requires two operations in the processor core: an effective address computation (executed by the Integer ALU) and the memory operation (which requires use of Read or Write ports). To support dynamic memory disambiguation, dynamically scheduled processors use a structure such as a load/store queue (LSQ), which keeps track of memory operations that have been issued but have not yet committed. Specifically, load/store operations need the following special treatment:

   - (In the dispatch stage): A load/store instruction results in two operations being created – an effective address generation operation, and the real load/store operation. The effective address computation is allocated an entry in the RUU, while the load/store is allocated an entry in the LSQ. The input and output operands for the original load/store are split up among the address computation and memory
operations as described in the skeleton file, `sim-multfu.c`. To ensure that the two operations are dependent upon each other, a (fake) output from the address generation operation is used as input for the load/store operation.

Because of this dependence, the load/store operation needs to wait for the effective address computation to finish before it can be dispatched. The load/store operation is deemed complete only when the memory operation part of it has completed.

- (In the issue stage): Stores are treated different from loads (and other instructions) in that they are assumed to complete in effectively zero time. This is because, once the effective address computation is complete, all that is required for a store is to update the LSQ entry with the store address. The rationale here is that memory values can be forwarded to affected loads through the LSQ, so the results of the store are immediately available. Since a store in the LSQ does not actually write out its value to memory until the instruction is committed (see the commit stage comment below), no FU allocation is required for the store until that stage.

Note that the reason for writing the store to the LSQ in a real processor is to determine whether or not subsequent loads can be issued (dynamic memory disambiguation) and/or forwarding values to the loads (if addresses match). However, in the simulator being developed for this assignment, you don’t have to handle either of these cases: because of our assumptions of in-order issue and perfect cache behavior, the latencies of store and load instructions are such that earlier stores to the same address will commit before a load instruction. The LSQ will play a bigger role in Assignment 4.

- (In the commit stage): When attempting to retire an RUU instruction, you need to check if the RUU entry contains an effective address generation instruction. If it does, the entry cannot be retired unless the corresponding LSQ entry has also completed operation (for stores, this completion happens as soon as the instruction gets issued; for loads, the completion happens in the writeback stage after the operation latency introduced by the FU handling the load).

If the LSQ entry has completed and is a store, then this is the point when FU resources need to be allocated for the store. As noted earlier, it is only in the commit stage that a store is written out to memory (this choice has been made so as to allow the processor core to conveniently be extended to handle speculation). Allocation of the FU unit for the store in the commit stage follows the same pattern as for other instructions in the issue stage.

Finally, retiring a memory operation ends up freeing both the RUU and LSQ entries.

**Sample Test Code and Sample Output**

The code distribution for this assignment is available at

`/griffin.b/faculty/vijayk/CompArch/assign3.tar`.

1. **Assembly Code Programs.** Two small sample assembly code programs: `fu.S`, and `loop3.S` have been provided as sample tests for you to use during your simulator development. To compile these, simply use `./griffin.b/faculty/vijayk/CompArch/bin/ssbig-na-trix-gcc` with the `-nostdlib` flag. The flag prevents the C standard library from being compiled into your code, thus limiting your instruction count to the number of instructions in your assembly code file (makes it easier to assess whether your cycle count is correct).

2. **Sample output.** Detailed stage-by-stage verbose output is provided for both programs. The output format is as described in the handout for Assignment 1, with the exception of the stage names: “fetch”, “dispa”, “issue”, “write”, and “commi”. 

   3
I have also modified the awk script from Assignment 2 (thanks to Geert Bosch for providing this), pipe2.script, which allows one to visually see the instruction flow through the different pipeline stages. Invoke the script as follows:

```
sh pipe2.script < fu.output > fu.out.pipe
```

**Submission Instructions**

Please send e-mail to the instructor (vijayk@cs.nyu.edu) by the due date attaching a tar file containing the following pieces: (1) your simulator source code (this should be just the file `sim-multfu.c`); (2) output generated by your simulator on the above test programs; (3) a brief README file describing your work and any outstanding problems.