In this assignment, you will extend the single-issue, in-order processor with multiple functional units that you constructed as part of Assignment 3 in three ways: (1) dynamic scheduling; (2) support for multiple instruction handling/cycle in the processor core; and (3) fetching of multiple instructions per cycle.

Background

For this exercise, we will use as a baseline the processor core architecture introduced in Assignment 3, which consists of Instruction Fetch, Dispatch, Issue, Writeback, and Commit stages that interact with multiple functional units (FUs) using a register update unit (RUU) and a load/store queue (LSQ). The architecture is similar to that introduced in class for implementing Tomasulo-like algorithms for dynamic scheduling. Please refer to the Assignment 3 handout for details.

As in Assignment 3, 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. As earlier, 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 in the Assignment 3 handout

The new features being introduced in this assignment include dynamic scheduling (allowing instructions whose register dependencies have been satisfied to be issued to functional units ahead of those that are awaiting completion of other operations), and support for handling multiple instructions per cycle. The latter feature is broken up into two parts: The first of these handles the situation where the Instruction Fetch stage is still fetching only one instruction per cycle, but the rest of the processor core is assumed to be limited only by the instruction dependencies and the number of functional units (i.e., unlike Assignment 3, we will model a processor that is capable of dispatching, issuing, writing back, and committing an arbitrary number of instructions per cycle). The second part adds the fetching of multiple instructions per cycle.

The Assignment

No code is being provided for this assignment. It is expected that starting from the skeleton code for Assignment 3, and the code you were supposed to implement as part of Assignment 3, you would be able to build in the new functionality required for this assignment. Any modifications to data structures and constants are identified in the following implementation notes.

The assignment consists of three parts, corresponding to the new features described above: (1) dynamic scheduling, (2) support for multiple instruction handling per cycle in the processor core; and (3) fetching of multiple instructions per cycle.

Dynamic Scheduling

The objective of this feature is to allow instructions whose register dependencies have been satisfied to be issued to FUs before earlier instructions that are still waiting on prior instructions to produce results. The mechanism

---

1 Note that in a real processor, there are limits beyond just the number of FUs that restrict how many instructions can be processed by each stage in a single cycle. Examples of such limits include the issue bandwidth, the number of data buses, register ports, etc. For simplicity, we shall restrict ourselves to only FU availability as the limiting factor.
you will be implementing is closely related to Tomasulo’s algorithm for dynamic scheduling that we discussed in class.

The overall structure of the implementation is as follows. If an instruction needs to read a register that is in the process of being written by an earlier instruction, then this instruction (the reservation station allocated for it) is associated with the reservation station of the producing instruction. This way, when the latter instruction has produced its result (in the writeback stage), the result can be communicated directly to all of the consuming instruction(s). This mechanism also has the nice side-effect of coping with WAW hazards, simply by allowing subsequent instructions to overwrite any previous producers of a register (since the data dependences have already been accounted for in the above association). These two effects are equivalent to the register renaming and data forwarding properties of reservation stations in Tomasulo’s algorithm.

Implementation notes:

1. Augment the reservation_station structure with an extra field, odep_list, which keeps track of the reservation stations that are interested in consuming the result that the current reservation station will produce (this field is an array of size MAX_ODEPS, corresponding to the number of different outputs that can be produced by the reservation station):

```c
struct reservation_station {
    /* instruction info */
    md_inst_t IR;
    enum md_opcode op;
    md_addr_t PC;
    md_addr_t addr; /* effective address for LD/STs */
    int will_exit;

    /* RS info */
    INST_TAG_TYPE tag; /* reservation station tag: increment to invalidate RS */
    INST_SEQ_TYPE seq; /* instruction sequence, used to sort the ready list and tag instruction */
    int in_LSQ; /* non-zero if op is in LSQ */
    int ea_comp; /* non-zero if op is an addr comp */

    /* instruction status */
    int queued; /* operands ready and queued */
    int issued; /* operation is/was executing */
    int completed; /* operation has completed execution */

    /* output dependent links */
    int onames[MAX_ODEPS]; /* output logical names */
    int odep_ready[MAX_ODEPS]; /* output operand ready? */

    struct RS_link *odep_list[MAX_ODEPS]; /* list of dependent reservation stations */

    /* input dependent links */
    int inames[MAX_IDEPS]; /* input logical names */
    int idep_ready[MAX_IDEPS]; /* input operand ready? */
};
```
2. Extend the definition of the `RS_link` structure so that you can keep track of which input of the reservation station being added to the odep_list structure above is dependent on the result being produced:

```c
struct RS_link {
    struct RS_link *next;
    struct reservation_station *rs; /* referenced RS */
    INST_TAG_TYPE tag; /* instr tag */
    union {
        tick_t when; /* time stamp of entry (for eventq) */
        INST_SEQ_TYPE seq; /* instr seq no. (for readyq) */
        int opnum; /* output dependence */
    } x;
};
```

3. Disable any code you might have added as part of your solution for Assignment 3 to ensure that a new instruction cannot be dispatched while a prior instruction is waiting for its dependencies to get resolved. Dynamic scheduling requires that it be possible to issue later instructions ahead of prior blocked ones.

4. Modify your implementations of the `install_idep`, `install_odep`, and `operands_ready` functions to implement the result forwarding and register renaming scheme discussed above. Basically, the current reservation station can be queued onto the odep_list(s) of any reservation stations that it needs to wait on to produce a result (this can be done as part of the `install_idep` function), and the reservation station should be identified as the current producer of its output register(s) (this functionality can be implemented in the `install_odep` function). `operands_ready` is a boolean function as earlier, which indicates whether or not the register dependencies of the instruction have been satisfied.

5. As part of the `writeback` stage, your implementation should propagate the results (logically) to all consuming reservation stations that have been queued onto the odep_list structure. If this propagation ends up satisfying the register dependencies of an instruction, then that instruction can be dispatched (i.e., put into the ready queue). Note that memory instructions are given special treatment as described in Assignment 3, with the modification described below.

6. In Assignment 3, because of in-order issue, we could schedule memory operations (only relevant for loads, because stores are handled in the commit stage) without checking to see if there were any conflicting operations. The latter refers to the case where a load might need to be prevented from executing because there is an earlier store instruction intended for the same memory address, which in turn is waiting on another (long-running) instruction to produce its result. In such cases, the hardware typically checks a ready load against pending memory operations to look for such conflicts, in a mechanism called dynamic memory disambiguation.

Dynamic memory disambiguation becomes an issue in this assignment because of dynamic scheduling. We might have an earlier store instruction blocked, either because its address is unresolved (i.e., the effective address computation is pending), or because the store value is still being computed. To ensure that we respect such memory conflicts, we shall implement the following scheme. The processor core is extended with an extra stage, `lsq_refresh`, whose job it is to inspect the load/store queue and dispatch any ready load instructions (put them into the ready queue). The invocation order of stages during each simulation loop is as follows:

```c
commit();
release_fu();
writeback();
lsq_refresh();
issue();
dispatch();
fetch();
```
As part of lsq_refresh, examine the load/store queue starting from the oldest instruction, keeping track of two kinds of conflicts: (1) a store instruction, whose address is yet to be resolved potentially has a conflict with all subsequent loads, i.e., none of them can be dispatched; (2) a store instruction whose address has been resolved but which is waiting for its data to become available has a conflict with subsequent load instructions to the same address. Any load instruction that has satisfied its register dependencies AND does not conflict with an earlier store instruction can be put into the ready queue.

Note that your simulated processor should be only handling 1 instruction per cycle in each of the stages. The only exception to this rule is in the writeback stage, when all instructions that become ready can be inserted into the ready queue. Because of the order in which the different stages are visited, you will observe the following effect: the writeback stage can insert an instruction into the ready queue, which is picked up by the issue stage in the same cycle. This is the desired behavior.

**Support for Multiple Instruction Handling Per Cycle in the Processor Core**

To observe the effect of additional processor core resources, in this part of the assignment, we shall build a simulator of a processor that still fetches a single instruction per cycle, but is capable of dispatching, issuing, writing back, and committing an arbitrary number of instructions per cycle, limited only by the number of functional units.

**Implementation notes:**

1. Implementing the above should be relatively straightforward for the dispatch, lsq_refresh, writeback, and commit stages: you would simply continue the operation of the stage until there were no more instructions that were ready to proceed with the stage.

2. The implementation of this functionality for this issue stage is slightly more complicated. This is because the issue stage needs to allocate a functional unit for the instruction before the latter can be considered issued. It is therefore possible that an earlier instruction in the ready queue (which is maintained in program order) may not have any available FUs in a given cycle, while a subsequent instruction might. In this case, your implementation should allow the subsequent instruction to be issued, while holding back the earlier one.

You can choose to implement this functionality either by allowing instructions to be dequeued from the middle of the ready queue, or more simply, by enqueuing instructions that cannot be immediately issued onto a temporary queue and reinitializing the ready queue with the contents of this queue (just setting the pointer should suffice). Note that if you choose the second implementation above, you may need to supply different implementations for the readyq_pop and readyq_enqueue functions to make sure that the verbose message printing works as originally intended.

**Multiple Fetches Per Cycle**

This should be really trivial if you have the other two parts of the assignment working.

**Implementation notes:**

1. Modify the FETCH_QUEUE_SIZE macro as below to allow up to 4 instructions to be fetched in a cycle:

   ```
   #define FETCH_QUEUE_SIZE (4)
   ```

---

2 Note that real processors implement forwarding from the load/store queue as well as allow a subsequent store to the same address to shadow a previous store (thus possibly removing the conflict for subsequent loads if the later store instruction has resolved its data). For simplicity, we shall ignore these optimizations in this assignment.
2. Modify the **fetch** stage so that you fetch as many instructions every cycle as are the number of empty slots in the **IF/ID** queue.

   Note that this implies you might fetch along the wrong path upon encountering branch instructions, but the implementation of perfect branch prediction outlined in the skeleton code supplied in Assignment 3 should already handle this case (by squashing all instructions in the fetch queue). You will need to handle the case where instructions might be fetched beyond the **exit syscall** instruction: this situation can be dealt with similar to the mispredicted branch case above.

---

**Sample Test Code and Sample Output**

The code distribution for this assignment is available at

```
/griffin.b/faculty/vijayk/CompArch/assign4.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 (these programs are the same as that supplied for Assignment 3). 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 for each of the three features above. 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”.

   I have also modified the awk script from Assignment 3 (thanks to Geert Bosch for providing this), *pipe3.script*, which allows one to visually see the instruction flow through the different pipeline stages. The script has been modified to handle printing of multiple instructions per stage per cycle. Invoke the script as follows:

   ```
   sh pipe3.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-dynissue.c*); (2) output generated by your simulator on the above test programs; (3) a brief README file describing your work and any outstanding problems.