ENEE 646: Digital Computer Design

Fall 2016 **Handout #12**

Programming Assignment #2 Due: Oct 17, 11:59pm

Out-of-Order MIPS Pipeline

In this assignment, you will build a sophisticated out-of-order processor model based on Tomasulo's Algorithm. Your processor model will support register renaming, out-of-order issue, memory disambiguation, and in-order commit. You will leverage much of the code provided for programming assignment #1, especially to implement the multi-cycle functional units, but you will add a lot of new code to implement the Tomasulo structures.

1 Files

The files for this assignment are available for download from http://www.ece.umd.edu/class/enee646/prog2. This time, there are 15 files: `main.c`, `pipeline.c`, `pipeline.h`, `fu.c`, `fu.h`, `output.c`, `output.h`, `Makefile`, `ooo_pipe.fu`, `simple.ooo.out`, `vect.ooo.out`, `newton.ooo.out`, `cos.ooo.out`, `vect.unroll.s`, and `vect.unroll.ooo.out`.

The first 8 files are the same as the corresponding files given for the in-order processor model, except a few additions have been made that are necessary for the out-of-order model. `ooo_pipe.fu` provides the functional unit configuration for this assignment, similar to `io_pipe.fu` from programming assignment #1. Finally, the remaining 6 files contain the output for the 4 benchmarks from programming assignment #1 on a properly functioning out-of-order processor simulator. Note, an additional benchmark has been included, `vect.unroll`. For this benchmark, both the source code and output are provided.

2 ISA and Assembler

In this programming assignment, you will simulate the same MIPS ISA (integer and floating point) from programming assignment #1. Hence, you should use the same assembler from the first assignment to create binaries for this programming assignment (*e.g.*, to assemble `vect.unroll.s`). See Sections 2 and 3 of handout #8 for details about the MIPS ISA and assembler.

3 Tomasulo Model

Your assignment is to create a cycle-accurate simulator of a Tomasulo-like machine that supports out-of-order issue and execution, and in-order commit. Your model will leverage many of the

features from programming assignment #1 (for example, the functional unit modules), so much of the experience you gained from completing that assignment will be useful here.

Like the first programming assignment, we have provided pipeline register data structures in `pipeline.h`. This time, the `state_t` processor structure, shown below, is more complex, and models a pipeline with 7 types of pipeline stages: fetch, dispatch, issue, memory_disambiguation, execute, writeback, and commit (see `pipeline.c`). Your machine will process a single instruction in the fetch, dispatch, issue, memory_disambiguation, and commit stages every cycle, but will allow a user-definable number of instructions to writeback per cycle (see below). Although your machine will address data hazards using out-of-order execution, it will still stall control hazards similar to programming assignment #1. In the remainder of this section, we give a summary of this processor structure. In Section 4, we will discuss how you should implement each pipeline stage in greater detail.

```
typedef struct _state_t {
  /* memory */
  unsigned char mem[MAXMEMORY];

  /* register files */
  rf_int_t rf_int;
  rf_fp_t rf_fp;

  /* pipeline registers */
  unsigned long pc;
  if_id_t if_id;

  IQ_t *IQ;
  int IQ_head, IQ_tail;
  CQ_t *CQ;
  int CQ_head, CQ_tail;
  ROB_t *ROB;
  int ROB_head, ROB_tail;

  fu_int_t *fu_int_list;
  fu_mem_t *fu_mem_list;
  fu_fp_t *fu_add_list;
  fu_fp_t *fu_mult_list;
  fu_fp_t *fu_div_list;
  wb_port_int_t *wb_port_int;
  int wb_port_int_num;
  wb_port_fp_t *wb_port_fp;
  int wb_port_fp_num;
  int branch_tag;

  int fetch_lock;
} state_t;
```

The fetch stage contains the program counter, "pc," and fetches an instruction each cycle into the "if_id" pipeline register. The dispatch stage moves an instruction from the "if_id" register into the instruction queue ("IQ") and reorder buffer ("ROB"), and possibly the conflict queue ("CQ"). For

all non-memory instructions, dispatch places the instruction in the "IQ" and "ROB" only. For memory instructions (*e.g.*, load and store), dispatch places the instruction in the "CQ" as well. Memory instructions are effectively split into two instructions: one that computes the effective address (a register value plus an immediate), and another that uses the computed effective address to access memory. The effective address computation part is placed in the "IQ," and the memory access part is placed in the "CQ." All three queues are managed as circular queues, each with their own "head" and "tail" pointers in the `state_t` structure to track where instructions should enter and leave. In addition to moving instructions from the "if_id" register to the queues, the dispatch stage also performs register renaming. We have provided tag fields in both the integer and floating point register files for this purpose (see definitions for the `rf_int_t` and `rf_fp_t` structures in `pipeline.h`).

The issue stage scans the "IQ" every cycle, searching for ready instructions to issue into the functional units. An "IQ" entry is ready for issue when all of its input operands are available. In this way, the "IQ" serves the same purpose as reservation stations, but it is a centralized structure with all instructions in the same queue (including both integer and floating point). The memory_disambiguation stage scans the "CQ" every cycle, searching for ready memory operations to issue into the memory functional units. A "CQ" entry is ready for issue when its corresponding "IQ" entry has completed (*i.e.*, the effective address has been computed), and when it has been disambiguated against all other memory operations in the "CQ." Once issued, instructions from the "IQ" and "CQ" perform one or more execute stages in the appropriate functional units. The functional unit types are identical to the ones from programming assignment #1 (see Section 4.2.2 in Handout #8), except there is an additional type, MEM. MEM functional units execute the memory access part of load and store instructions; instructions in the "CQ" issue into these functional units. Instructions for all other functional units are issued from the "IQ."

When an instruction completes in one of the functional units, it must arbitrate for one of the writeback ports. (Unlike programming assignment #1 where writeback structural hazards were resolved in the decode stage at the front of the pipeline, you will resolve writeback structural hazards by stalling instructions in the functional units for the Tomasulo machine). Integer and floating point instructions reserve a writeback port by allocating an entry in the `wb_port_int` or `wb_port_fp` arrays, respectively. The number of writeback ports is configurable at the command line; the simulator accepts the "-wbpi" and "-wbpf" flags to specify the number of integer and floating point writeback ports, respectively. Although you can set these parameters freely, the solution *.ooo.out files provided all assume 2 integer and 2 floating point writeback ports. Writeback port arbitration occurs as part of the execute stage (essentially, it's the last execute stage in each functional unit). Control instructions have their own writeback slot, the `branch_tag` field. Upon completion, a control instruction moves into this field. Since we stall all control instructions, there can only be one control instruction in-flight at any moment in time, so there is no need for arbitration of the `branch_tag` slot.

The writeback stage scans the `wb_port_int` and `wb_port_fp` arrays and the `branch_tag` field for completed instructions, and writes their results into the "ROB." The "ROB" stores these completed instruction results for forwarding to subsequent instructions as they enter the "IQ" and "CQ" structures from the dispatch stage. Finally, the commit stage commits the oldest instruction in the "ROB" (*i.e.*, the instruction at the head of the "ROB") into the register file or memory (the latter is for store instructions).

4 Implementing the Pipeline Stages

Having summarized the `state_t` processor structure, we now describe in greater detail the functions performed by each pipeline stage. Use the following description to guide your implementation of each pipeline stage, and ultimately, of the entire Tomasulo machine.

4.1 Fetch

The fetch stage in the Tomasulo machine is no different from the fetch stage in the in-order machine from programming assignment #1. Also, since we will stall all control instructions, we should only perform fetch as long as there are no in-flight control instructions. As in programming assignment #1, this is indicated by the "fetch_lock" flag equal to "TRUE" (see the main simulator loop in `main.c`). The "fetch_lock" flag is set whenever a control instruction is dispatched, as we will describe in Section 4.2.

4.2 Dispatch

The dispatch stage takes the most recently fetched instruction from the "if_id" pipeline register, and inserts the instruction into the instruction queues. The only instruction that should not be moved into the queues is the NOP instruction (see Section 4.3.4 in Handout #8). Since this instruction does nothing, it should not be processed further by the pipeline to save execution bandwidth in subsequent stages. For all other instructions, the dispatch stage should try to move the instruction into the "IQ," "CQ," and "ROB" structures, as appropriate. In the following, we describe processing of instructions for each of the 3 queues.

Instructions should be inserted into the "ROB" at the tail, specified by the "ROB_tail" variable. Each "ROB" entry has 4 fields: `instr`, `completed`, `result`, and `target`. The `instr` field should be set to the instruction itself. The `completed` field is a flag that indicates whether the instruction has finished execution in the appropriate functional unit, or whether the instruction is still executing. Upon dispatch, all instructions should have a `completed` field equal to "FALSE," except for the `halt` instruction. The `halt` instruction does not need to execute in a functional unit, so it is completed (and its `completed` field set to "TRUE") immediately upon dispatch. The `result` field stores the result of completed instructions after execution and writeback, so it is uninitialized during the dispatch stage. The `target` field is used for control and memory instructions, and stores the instruction's target address or effective address, respectively. Again, this value is not known until after execution, so it is uninitialized during dispatch. Note, for control instructions, the `result` field should store either the link address for jump-and-link instructions, or the condition (taken or not taken) for conditional branches. After "ROB" insertion, the "ROB_tail" variable should be incremented to reflect the newly inserted instruction. Since the "ROB" is a circular queue, increment the "ROB_tail" variable modulo the size of the "ROB," "ROB_SIZE" (*i.e.*, if the tail of the "ROB" is at index "ROB_SIZE" - 1 prior to insertion, then the new "ROB_tail" should point to index 0).

Instructions should be inserted into the "IQ" at the tail, specified by the "IQ_tail" variable.

All instructions, except for NOP and HALT, should be inserted into the "IQ." Each "IQ" entry has several fields. The `instr` and `pc` fields should be set to the instruction and its corresponding PC, respectively. The `issued` field is a flag that indicates whether or not the instruction has been issued; this flag should be initialized to "FALSE." The `ROB_index` field should be set to the location in the "ROB" where the instruction was inserted. Lastly, the `tag1`/`operand1` and `tag2`/`operand2` fields are the register tags and operand value for the first and second source operands, respectively.

The tag fields in the "IQ" along with the tag fields in the integer and floating point register files facilitate register renaming. We recommend the following tag convention: a tag value of "-1" indicates the value is ready and present, while a tag value different from "-1" indicates the value is being computed by an in-flight instruction. In the latter case, the tag value should be in the range 0 to "ROB_SIZE" - 1, and indicates the *index in the "ROB" where the in-flight instruction resides*. Using this convention, register renaming can proceed as follows. First, check the desired register. If the value is present (*i.e.*, the register's tag == -1), immediately read the register value into the "operand" field of the "IQ," and set the appropriate tag in the "IQ" to "-1" (*i.e.*, the operand is ready). If, however, the register value is not present, then there is an in-flight instruction producing the desired value. In this case, check the in-flight instruction's `completed` flag in the "ROB" (the register file's tag value points to the ROB entry you should check). It is possible for this instruction to have completed, but not yet committed. If so, the result is available in the "ROB," and should be copied into the "operand" field of the "IQ" (and the corresponding tag field set to "-1"). If the in-flight instruction has not completed, then no operand is available. Set the tag field in the "IQ" to the ROB index of the in-flight instruction. This will stall the "IQ" entry until the in-flight instruction completes and forwards the missing register value. After renaming the source registers, the destination register should be renamed. This is done by setting the destination register's tag in the register file to the ROB index of the dispatched instruction. After register renaming and "IQ" insertion, increment the "IQ_tail" variable modulo "IQ_SIZE."

Be careful when renaming registers–there are many cases to handle. Some instructions have two source register operands, while others only have one. For example, instructions that use immediates only read one source register value; the other operand is an immediate that can be extracted from the instruction (use the `FIELD_IMM` or `FIELD_IMMU` macros). In this case, the immediate operand should always be copied to the "operand" field in the "IQ" with tag == -1 (since it's always ready). Also, some instructions don't produce values. For example, jumps and branches don't write destination registers, and so they should not create a new register name in the register file tags (though an exception to this is jump-and-link, which writes R31). Lastly, be sure to handle R0 correctly. A write to R0 does nothing, so you should never rename R0. R0's tag in the register file should always be "-1."

For all non-memory instructions, dispatch occurs to the "IQ" and "ROB" only. For loads and stores, dispatch also involves the "CQ." As described in Section 3, loads and stores are split into 2 instructions: an effective address computation and a memory access operation. The effective address computation resembles an ADDI instruction (register value plus immediate), and is inserted into the "IQ" as described above. In contrast, the memory access operation is inserted into the "CQ." "CQ" entries have the same fields as "IQ" entries, with the exception that there is no `pc` field, and there is an extra `store` field that is "TRUE" for stores and "FALSE" for loads. Like "IQ" entries, each "CQ" entry has two tag/operand pairs. Use tag1/operand1 to link the "CQ" entry to the corresponding "IQ" entry that computes the effective address. In other words, `tag1` is not

ready until the corresponding "IQ" entry produces the desired effective address and forwards it to the `operand1` field. This will stall loads and stores in the "CQ" until their effective address becomes available. Use tag2/operand2 to handle the second source register for store instructions (loads do not need tag2/operand2; you can set tag2 == -1 for loads). In other words, you should send the register value to store to memory to the "CQ," not the "IQ." This allows store instructions to stall for the second source register operand in parallel with issuing the effective address computation. After "CQ" insertion for loads and stores, increment the "CQ_tail" variable modulo "CQ_SIZE."

As described in Section 4.1, we stall fetch after dispatching a control instruction. So, set the `fetch_lock` variable to TRUE whenever you encounter a jump or branch in the dispatch stage. Also, dispatching a `halt` instruction should disable fetch for the remainder of the simulation (the simulation will terminate when the `halt` instruction commits). Finally, dispatch should fail and cause a stall if any of the queues required by the instruction ("IQ," "CQ," or "ROB") are full. Since the instruction queues are circular, the queue-full condition occurs when the head of the queue is located one entry before the tail of the queue modulo the queue size.

4.3 Issue

The issue stage scans the "IQ" for ready instructions, and issues them into available functional units. Starting from the head of the "IQ," look for the first instruction that has not been issued *and* all necessary operands are ready (*e.g.*, tag == -1). Once such an instruction has been found, try to issue the instruction into a functional unit of the type required by the instruction. Use the `issue_fu_int` or `issue_fu_fp` routines for this purpose. If issue is unsuccessful due to insufficient functional unit resources, look deeper into the "IQ" for another instruction that can issue. Repeat until an instruction is successfully issued, or the tail of the "IQ" is reached. If an instruction is successfully issued, be sure to set its `issued` bit to "TRUE." Issue at most one instruction per cycle.

Recall from programming assignment #1 there are many possible places in the pipeline where functional execution of instructions can occur (see Section 4.3.3 in Handout #8). The same applies to your Tomasulo machine. However, for this assignment, we recommend that you perform functional execution of each instruction in the issue stage when the instruction is successfully issued into a functional unit. You should place the instruction's result in the "ROB," but of course, you should not set the `completed` bit until the instruction enters the writeback stage. Granted, this means instruction results enter the "ROB" too early. But our output routines never print the result fields of the "ROB," so this timing discrepancy will not be visible to the outside world. (Note, "IQ" entries for loads and stores only compute the effective address–they do not perform the memory access part–so, only the effective address result should be placed in the "ROB" which resides in the "target" field).

At the end of the issue stage, remove any issued instructions by incrementing the "IQ_head" variable (the increment should be performed modulo "IQ_SIZE"). Any number of issued instructions can be removed each cycle, but removal must start from the head of the "IQ," proceed down the "IQ" sequentially, and stop once an unissued instruction is encountered. In other words, you should never remove an instruction from the "IQ" unless all older instructions have also been removed.

4.4 Memory_Disambiguation

The memory_disambiguation stage scans the "CQ" for ready memory operations, checks for conflicts, and issues the memory operations if no conflicts exist. Scan the "CQ" starting from the head, indicated by the "CQ_head" variable. As you scan, once you encounter a store without its effective address, *i.e.*, tag1 != -1 (see Section 4.2), then it's impossible to disambiguate any subsequent memory operations. In this case, memory disambiguation fails, and no memory operations can issue that cycle. As you scan the "CQ," if you encounter an unissued store with both operands ready (*i.e.*, tag1 == tag2 == -1), then issue the store immediately. Stores do not need to allocate a functional unit, and can always issue when their operands are ready (more about this in a moment). As you scan the "CQ," if you encounter an unissued load with its effective address ready (*i.e.*, tag1 == -1), check for conflicts with an older store. Rescan the "CQ" from the head to see if an earlier unissued store is writing the same location. If a conflict is detected, the load cannot issue. In this case, keep scanning through the "CQ" for another instruction to issue. If no conflict is detected, try to issue the load into a functional unit by calling `issue_fu_mem`. If no MEM functional unit is available, then you have a structural hazard; memory disambiguation fails, and you should not continue trying to issue an instruction. If a MEM functional unit is available, then the load issues successfully. When issuing a load or a store from the "CQ," be sure to set the corresponding `issued` bit to "TRUE."

As described in Section 4.3, we recommend that you perform functional execution of each instruction during issue; the same applies to memory operations issued from the "CQ." For stores, there is nothing to execute when issuing. This is because stores are not allowed to write memory until the commit stage to maintain precise processor state. Prior to commit, stores do nothing except possibly forward their store values to waiting loads later in the pipeline. So, stores effectively execute in zero cycles, and complete as soon as their operands are ready. Hence, when issuing a store in the memory_disambiguation stage, immediately set the `completed` bit in the corresponding "ROB" entry to "TRUE," and place the store value in the `result` field of the "ROB" entry. For loads, functional execution normally involves accessing the load value from memory. However, before going to memory to perform the load, you should first check the "ROB" to see if an earlier completed store is writing the same memory location. If so, you should forward the store value to the load instead of loading the value from memory. If there is no such store in the "ROB," then simply access memory to acquire the load value. The load value should be written into the appropriate "ROB" entry's `result` field.

Issue at most one memory operation from the "CQ" every cycle. At the end of the memory_disambiguation stage, remove any issued memory operations by incrementing the "CQ_head" variable modulo the "CQ_SIZE." Like the "IQ" described in Section 4.3, any number of issued memory operaions can be removed each cycle, but removal must start from the head of the "CQ," proceed down the "CQ" sequentially, and stop once an unissued instruction is encountered.

4.5 Execute

The execute stage advances all instructions in the functional units by 1 cycle. Similar to programming assignment #1 (see Section 4.2.2), we have provided functions to simulate the functional units

in `fu.c`: `advance_fu_int`, `advance_fu_fp`, and `advance_fu_mem`. Simply call these functions once for each functional unit list in the `state_t` processor structure.

4.6 Writeback

As described in Section 3, instructions arbitrate for writeback ports when they complete execution, and reserve an entry in either the `wb_port_int` or `wb_port_fp` arrays. This arbitration is already implemented for you in the `advance_fu_int` and `advance_fu_fp` routines. All you need to do in the writeback stage is to check these writeback arrays for instructions that are ready to write back, and perform the writeback for the instruction. (You can tell a particular writeback entry is non-empty if its tag field is not equal to -1).

First, writeback the integer instructions from `wb_port_int`. For each integer instruction, find its corresponding entry in the "ROB" (the tag field in `wb_port_int` specifies the index in the "ROB" where the instruction resides). Set the completed flag to TRUE. Remember, the result of the instruction was already computed when you issued the instruction (see Section 4.3); you are only marking the completion of the instruction. One exception is the ADDI portion of a load or store instruction. The writeback of this part should not set the completed flag to TRUE because only the effective address has completed; you should set the completed flag to TRUE for loads and stores only when the entire memory operation has finished. Next, broadcast the result to the IQ and CQ, matching the tag against any waiting instructions. If a match is found, deposit the result of the instruction into the appropriate operand field in the IQ or CQ, and set the corresponding tag field to -1, signaling to the issue stage that this operand is now ready.

Second, writeback the floating point instructions from `wb_port_fp`. This should proceed in the same fashion as the integer instructions. The only difference is that a floating point result should never match the tag1 field in the CQ. The tag1 field for CQ entries is reserved for effective address results which can never be produced by a floating point instruction (they can only be produced by the ADDI portion of a load or store instruction).

Finally, if a branch instruction completes, it will occupy the branch_tag field. If this field is non-empty (*i.e.*, not equal to -1), then you should "writeback" the branch. This involves setting the completed bit for the branch instruction at the corresponding "ROB" entry to TRUE. Also, if the branch is taken, then the target of the branch should be copied into the program counter, and the instruction in the "if_id" pipeline register should be squashed by replacing it with a NOP instruction (see Section 4.3.4 in Handout #8). If the branch is not taken, then no action needs to be done in the fetch stage. After the branch writes back, you should set the `fetch_lock` variable to FALSE, thus enabling fetching again. Normally, branches do not need to writeback to the register file. The only exception is jump-and-link instructions since they must write the link address to R31. Upon completion of execute, jump-and-link instructions will be placed in the branch_tag field similar to other control instructions, but they will also arbitrate for a slot in the `wb_port_int` so that their link address can be written into the register file.

4.7 Commit

The commit stage examines the instruction at the head of the "ROB." If this instruction has not yet completed yet, commit fails for the current cycle, and you retry on the following cycle. Otherwise, commit occurs for the instruction at the head of the "ROB."

For instructions that produce register results, write the result into the register file (either integer or floating point). Next, check to see if the committing instruction's tag (*i.e.*, "ROB" index) equals the register's current tag. If so, set the corresponding tag in the register file to -1. Note, it's possible the tags won't match if another instruction has "re-renamed" the register. In that case, you still write the result into the register file, but you don't make the tag ready.

Loads commit their results into the register file, but stores do not. Instead, stores should actually perform their operation to memory. You should "issue" the store to memory by calling `issue_fu_mem`. If this fails due to a structural hazard, then you cannot commit the store on the current cycle, and you retry on the following cycle. Otherwise, you can issue the store, and copy the value to memory. Note, go ahead and copy the value to memory right away on the cycle of commit. In reality, this is too early (the value should actually go to memory when `fu.c` simulates the memory cycle of the store), but for this assignment we'll perform the store on the cycle of commit.

Lastly, committing a HALT instruction terminates the simulation. Notice, NOPs should never commit since they are discarded at dispatch. At the end of the commit stage, remove the committed instruction by incrementing the "ROB_head" variable modulo the "ROB_SIZE."

5 Provided Code

Similar to programming assignment #1, a fair amount of code has been provided for you in main.c, fu.c and output.c. This code is very similar to the corresponding modules from programming assignment #1 (see Section 4.2 of Handout #8).

As before, fu.c implements the integer/floating point functional units and simulates their execute stages. There are a few differences compared to the previous fu.c module. First, a new memory functional unit, "MEM," has been added. This functional unit executes the memory part of load and store instructions, so you will issue memory operations into this functional unit from the "CQ." (In other words, loads and stores do not execute entirely in the INT functional units anymore; instead, only the ADDI part for effective address computation executes in INT units, while the memory part executes in MEM units). The same interface exists for the MEM functional unit as exists for the INT, ADD, MULT, and DIV functional units. Specifically, there are `fu_mem_read`, `issue_fu_mem`, and `advance_fu_mem` functions implemented in fu.c that serve the same purpose as the corresponding functions for the INT, ADD, MULT, and DIV instructions.

Second, as discussed earlier, the last execute stage of each functional unit type arbitrates for a writeback slot in the `wb_port_int` or `wb_port_fp` arrays. This is done by calling the `wb_int` and `wb_fp` functions for integer and floating point instructions, respectively, to perform the arbitration.

The only exceptions are branches, which don't need to arbitrate for the branch_tag field since there can only be one branch executing at a time (though jump-and-link instructions still need to arbitrate for a `wb_port_int` slot), and stores, which don't need to arbitrate because they don't perform any writeback (they write memory when they commit–see Section 4.7).

Lastly, fu.c no longer implements `fu_int_done` and `fu_fp_done`. Instead of waiting for all instructions to flush out of the execution units after decoding a halt like you did in programming assignment #1, we now can detect the halting condition as soon as the HALT instruction goes to commit.

In addition to fu.c, we have provided an output.c module that implements the `print_state` routine for the Tomasulo pipeline. As before, this routine dumps the state of memory, the register files, and the pipeline registers. In addition, it also dumps the state of the instruction queues, "IQ," "CQ," and "ROB." The dump for the instruction queues includes useful information for each instruction, like its index location in the queue, the status and values of tags and the issue status (for the "IQ" and "CQ"), and the completion status (for the "ROB").

6 Grading

We will grade your programming assignment by comparing the output of your simulator on each of the five benchmarks against the output files `simple.ooo.out`, `vect.ooo.out`, `vect.unroll.ooo.out`, `newton.ooo.out`, and `cos.ooo.out`. So, your goal should be to design your simulator such that you get the exact same output as the output files we've provided.