Homework 5 - CIS501 Fall 2010

Instructor:	Prof. Milo Martin
Due:	Thursday, December 8th (at the **start** of class)
Instructions:	This is an *individual* work assignment. Sharing of answers or code is strictly prohibited. For the short answer questions, **Show your work** to document how you came up with your answers.
Extensions:	For those of you that have not used up your allotment of extensions, you may turn in the assignment by noon on Monday, December 12th.

Overview

This assignment explores the operation of out-of-order execution (dynamic scheduling) and its effectiveness in extracting instruction-level-parallelism. As a fully detailed simulation of such a core is beyond what could be feasibly done for an assignment, the model of the processor core described in this assignment is approximate in many ways (failing to capture many second-order effects), but it does capture the most important first-order effects of instruction execution.

This assignment uses the same traces as previous assignments (see Trace Format document).

As before, you'll write a program that reads in the trace and simulates different processor core configurations. You're welcome to write the program in whatever language you like; although we are going to ask for you to turn in your source code, the end result of this assignment are the experimental results (the program source code is just a means to the end).

Step #1: Register Renaming

The first step is applying register renaming to the instruction stream. You will need two data structures for this:

- **Map table**: to map an *architectural register* to a *physical register* you'll need an array (or "vector") with one entry for each architectural register. Assume the set of architectural registers is from zero to 49 (so you'll need an array of size 50). At the start, initialize the table so that architectural register *n* is mapped to physical register *n*.
- **FIFO free list**: a queue of physical register names. The operations needed to be supported are: (1) getting a free register from the head of the queue, (2) putting a newly deallocated register at the tail of the queue, and (3) querying the number of registers that are free. You may wish to use an existing queue data structure implementation (such as "ArrayDeque" in Java or "deque" in C++'s STL).

Renaming Algorithm

The renaming algorithm:

```
For each valid source register:
  microop.phys_reg_source[i] = maptable[microop.arch_reg_source[i]]

For each valid destination register:
  microop.phys_reg_to_free[i] = maptable[microop.arch_reg_destination[i]]
  new_reg = freelist.dequeue()
  maptable[microop.arch_reg_destination[i]] = new_reg
  microop.phys_reg_destination[i] = new_reg
```

At commit:

```
for each valid destination register:
  freelist.enqueue(microop.phys_reg_to_free[i])
```

Handling flags. One wrinkle in dealing with x86 is that we need to handle the flags (condition codes). To avoid WAW and WAR hazards on the flags, you must rename them as well. To do this, we'll treat the flags like just another architectural

register by assigning it architectural register 49. Any instruction the reads (or write) the flags is then set to read (or write) architectural register 49. Thus, each micro-op has up to three source registers and up to two destination, all of which must be renamed according to the above algorithm.

Renaming Test Cases

To help you debug your renaming code, we've provided some renamed traces for you to compare your code with. Even without all the rest of the out-of-order machinery, you can generate a trace of the renamed instruction stream for a given number of physical registers. To generate a rename trace, instead of waiting to commit to free the registers, you can just free the register right after renaming it; the FIFO (first-in, first-out) nature of the queue will ensure the rename trace is the same.

The output rename trace has the following format:

```
43, r7 -> p49, r5 -> p7 [p5] | MOV LOAD
44, r5 -> p7, r44 -> p58 [p13], r49 -> p50 [p59] | CMP SUB_IMM
45, r49 -> p50 | J JMP_IMM
46, r45 -> p44 [p52] | CMP SAVE_PC
```

The first column is the micro-op count (the line number from the trace). This is followed by pairs of architectural registers and their mapping to physical registers. The destination registers also include the register that should be freed when they commit in square brackets. If an instruction doesn't have a valid architectural register, it doesn't display anything for that register. The register mappings are followed by the pipe ("|") symbol and then the macro-op and micro-op descriptors.

The renamed instruction trace for the sjeng-1K.trace.gz for 60 and 256 registers, respectively:

- sjeng-1K-rename-60-regs.output
- sjeng-1K-rename-256-regs.output

You can download all the traces at: sjeng-1K.outputs.tar.gz

Step #2: Dynamic Scheduling

Now that you've implemented register renaming in Step #1, the next step is to implement the pipeline simulation for out-of-order execution (dynamic scheduling) and explore its impact on instruction-level parallelism.

To simulate an (idealized) out-of-order processor core, we are going to model a reorder buffer (ROB) of finite size, but assume there are plenty of physical registers, instruction queue entries, load queue entries, store queue entries, and assume that an N-way superscalar machine can always fetch N instructions and execute any N instructions in a given cycle. All non-loads will have a single-cycle latency; loads have a four-cycle latency (three-cycle load-to-use penalty) and never miss in the cache. The pipeline is also simplified in that we are going to model a relatively short pipeline.

As before, we have generated test cases that allow you to check your simulator to make sure it is executing correctly (see below).

Reorder Buffer (ROB)

The key data structure for dynamic scheduling is the reorder buffer (ROB). It has fixed number of entries, with one entry for each micro-op than has been fetched and not yet committed. The ROB is managed as a queue. At instruction fetch, micro-ops are enqueued at the tail of the ROB. At instruction commit, micro-ops are dequeued from the head of the ROB.

Information for Each Micro-op

I suggest that you make each entry of the ROB an object (or struct) with all the information about each micro-ops:

- **SourceRegisters**. The input (source) physical registers (up to three of them)
- **RegisterReady**. Is the corresponding input register ready? (up to three registers)
- **RegisterDestinations**. The output (destination) physical registers (up to two of them)
- **isLoad**. Is the micro-op a load operation?
- **isStore**. Is the micro-op a store operation?
- **Address**. If the memory operation is a load or a store, this field specifies the memory location it accesses (note: this is not the PC of the instruction, but the address loaded or stored by the micro-op).
- **Sequence Number**. This records the "age" of the micro-op. For simplicity, this can be encoded as an

integer value as to how many micro-ops have been fetched thus far in the simulation. In reality, this would be encoded in fewer bits, using wrap-around arithmetic when comparing two micro-ops.
- **Issued**. Has this instruction been issued?

As you fetch, issue, finish execution, and commit each micro-op, you'll want to track when you do each. These wouldn't always exist in a real design, but we'll use them for various bookkeeping and for generating the debugging trace.

- **FetchCycle**. Cycle the micro-op was fetched.
- **IssueCycle**. Cycle the micro-op was issued.
- **DoneCycle**. Cycle the micro-op completed execution.
- **CommitCycle**. Cycle the micro-op committed.

You'll also want to include some information for the debugging traces:

- **MacroOp name**. The text string of the macro-op name.
- **MicroOp name**. The text string of the micro-op name.

Dynamic Scheduling Algorithm

To prevent an instruction from flowing through the pipeline in a single cycle, we perform the pipeline stages in reverse order.

1. Commit

During commit, we look at the micro-op at the head of the ROB and commit (dequeue) it if it has completed executing. We can commit at most N instructions per cycle:

```
for (i=0; i<N; i++):
  if (ROB.head().DoneCycle <= CurrentCycle):
    ROB.dequeue()
```

When generating the output trace, the micro-op should be printed to the debug output right before it commits.

2. Issue

During issue, we select the N oldest ready instructions to begin execution:

```
count = 0
foreach microop in ROB (iterating from the oldest to the youngest):
  if (microop.issued == false) and microop.isReady():
    microop.issued = true
    microop.issuedCycle = CurrentCycle
    microop.DoneCycle = CurrentCycle + latency

    foreach destination physical register in microop:
      set scoreboard[physical register] = latency

    count = count + 1
    if (count == N):
      break            // stop looking for instructions to execute
```

When is a Micro-op "Ready"

A micro-op is considered ready when all of its source (input) registers are ready (for now, ignore memory dependencies). To track which registers are ready or not, a real processor has a ready table (one bit per physical register) and also ready bits in each entry in the instruction queue, which are updated during instruction wakeup. Although we could model this sort of wakeup operation in the simulator, it is easier to use a register scoreboard that records how many cycles until a register is ready (just as we did for the pipeline simulator, except acting on physical registers rather than architectural registers).

Thus, the second key data structure is the "scoreboard" table. This table has one entry per physical register. Each entry is an integer that indicates how many cycle until the register will be ready. The value of zero indicates the register is ready. A negative value says the instruction that write that physical register is waiting to issue. A positive value indicates the number

of cycles until the instruction that writes that register will generate its output value. At the start of the simulation, initialize all physical registers as ready (value of zero).

To determine if a regiser is ready, your code should just check to see if the entry in the scoreboard is equal to zero. If all the source physical registers are ready, then the instruction is ready to issue:

```
bool isReady(MicroOp op):
  foreach physical_register in op's source physical registers:
    if physical_register is not ready
      return false
```

3. Fetch & Rename

At fetch, instruction are renamed and inserted in the ROB:

```
for (i=0; i<N; i++):
  if ROB.isFull():
    break;    // ROB is full

  1. Read the next micro-op from the trace
  2. Rename the micro-op (as per the previous assignment)
  3. Enqueue the micro-op into the ROB, filling in the fields as appropriate
  4. foreach valid destination physical register
       set the register as "not ready" by setting the correspond scoreboard entry
```

4. Advance to the Next Cycle

At the end of each cycle, we need to advance to the next cycle:

```
CurrentCycle = CurrentCycle + 1
foreach preg in Scoreboard:
  if (Scoreboard[preg]) > 0:
    Scoreboard[preg] = Scoreboard[preg] - 1;    // decrement
```

Dynamic Scheduling Test Cases

We've made several output debug traces available to help you debug and test your code. The debug output trace looks like:

```
1: 0 1 2 2, r13 -> p50 [p13] | SET ADD
2: 0 1 2 2, r49 -> p49, r13 -> p51 [p50] | SET ADD_IMM
3: 0 1 4 4, r5 -> p5, r45 -> p52 [p45] | CMP LOAD
4: 0 4 5 5, r45 -> p52, r3 -> p3, r44 -> p53 [p44], r49 -> p54 [p49] | CMP SUB
5: 1 2 5 5, r5 -> p5, r3 -> p55 [p3] | MOV LOAD
6: 1 2 3 5, r0 -> p56 [p0] | SET ADD
7: 1 5 6 6, r49 -> p54, r0 -> p57 [p56] | SET ADD_IMM
8: 1 2 3 6, r12 -> p58 [p12] | XOR ADD
9: 2 6 7 7, r13 -> p51, r0 -> p57, r13 -> p59 [p51], r49 -> p60 [p54] | OR OR
10: 2 3 4 7 | JMP JMP_IMM
11: 4 5 8 8, r3 -> p55, r0 -> p61 [p57] | MOV LOAD
12: 5 8 9 9, r0 -> p61, r0 -> p61, r44 -> p62 [p53], r49 -> p63 [p60] | TEST AND
13: 5 9 10 10, r49 -> p63 | J JMP_IMM
14: 5 8 11 11, r0 -> p61, r7 -> p64 [p7] | MOV LOAD
```

The first column is the number of the micro-op. The next four columns are the cycle in which the micro-op was: fetched, issued, completed execution, and committed. The remaining columns are the same as the output the rename trace described above.

We've provided multiple debugging outputs for each of the experiments below. To make it easier to debug, we've provided: (1) debug traces with just a few ROB entries and single-issue, and (2) slightly larger ROB and four-way issue, and (3) large ROB with full 8-wide exeuction. The number of physical registers also vary (but is always sufficient to not stall the pipeline) to test cases in which registers get reused. The names of the traces encode the exact settings of these parameters:

- sjeng-1K-regs=64-rob=4-width=1-exp=1.output

- sjeng-1K-regs=64-rob=4-width=1-exp=2.output
- sjeng-1K-regs=64-rob=4-width=1-exp=3.output
- sjeng-1K-regs=64-rob=4-width=1-exp=4.output
- sjeng-1K-regs=64-rob=4-width=1-exp=5.output
- sjeng-1K-regs=128-rob=8-width=4-exp=1.output
- sjeng-1K-regs=128-rob=8-width=4-exp=2.output
- sjeng-1K-regs=128-rob=8-width=4-exp=3.output
- sjeng-1K-regs=128-rob=8-width=4-exp=4.output
- sjeng-1K-regs=128-rob=8-width=4-exp=5.output
- sjeng-1K-regs=2048-rob=128-width=8-exp=1.output
- sjeng-1K-regs=2048-rob=128-width=8-exp=2.output
- sjeng-1K-regs=2048-rob=128-width=8-exp=3.output
- sjeng-1K-regs=2048-rob=128-width=8-exp=4.output
- sjeng-1K-regs=2048-rob=128-width=8-exp=5.output

You can download all the traces at: sjeng-1K.outputs.tar.gz

Experiments

The following experiments incrementally refine the simulation to less idealized and explores the impact on performance of those changes.

Experiment #1 - Unrealistic Best-Case ILP

The first experiments explores the amount of instruction level parallelism for various ROB sizes. This result is going to be overly optimistic, as this experiment is going to totally ignore memory dependencies and ignore branch prediction. As noted above, loads are multi-cycle instructions and all other instructions are single-cycle instructions. For these experiments, set N to 8 (the processor can fetch, rename, issue, execute, and commit up to eight micro-ops per cycle).

Vary the size of the ROB from 16 micro-ops to 1024 micro-ops in powers of two (16, 32, 64, 128, 256, 512, 1024). As we are focusing on the ROB size and not the size of the physical register file, just set the total physical registers to 2048 for all experiments (which is sufficient to avoid any problems with running out of free registers).

Run the simulations for each of these ROB sizes and record the micro-ops (instructions) per cycle (IPC). Plot the data as a line graph with the IPC on the y-axis and the **log of the ROB size** on the x-axis. In essense, the x-axis is a log scale. Without such a log scale, all the interesting data points are crunched to the far left of the graph and it is basically unreadable. This is basically the same thing we did with the branch predictor sizes and cache sizes, so this isn't anything new.

Experiment #2 - Conservative Memory Scheduling

In the above experiment, the simulation ignored memory dependencies completely. For this experiment, implement conservative memory scheduling. That is, in addition to the requirement that all source registers must be ready, a load can issue only when all prior stores have issued:

```
bool isReady(MicroOp op):
  foreach physical_register in op's source physical registers:
    if physical_register is not ready
      return false
  if op.isLoad:
    foreach op2 in ROB:
      if (op2.isStore) and (op2.SequenceNum < op.SequenceNum) and (op2.isNotDone(
        return false
  return true
```

An instruction is "Done" if it has issued (op.issued is true and DoneCycle is <= Current Cycle). Thus, an instruction being "NotDone" is just the negation of "Done".

Based on the above conservative memory scheduling, re-run the simulation for the same ROB sizes as the earlier experiment. Plot the data (IPC vs ROB size) on the **same** graph.

Experiment #3 - Perfect Memory Scheduling

Neither of the above two experiments are representative of state-of-the-art memory scheduling. In this experiment, we will use the fact that our trace already has memory addresses in it to implement perfect memory scheduling. Adjust the "isReady()" computation to include an addresses match:

```
bool isReady(MicroOp op):
  foreach physical_register in op's source physical registers:
    if physical_register is not ready
      return false
  if op.isLoad:
    foreach op2 in ROB:
      if (op2.isStore) and (op2.SequenceNum < op.SequenceNum) and (op2.isNotDone(
        if (op.Address == op2.Address):
          return false
  return true
```

The only addition to the above algorithm is the check that delays a load if and only if the earlier store is to the same address.

> **Note**
>
> We are actually still being a bit optimistic, because an 8-byte load might depend on a mis-aligned 8-byte store. As the addresses of these two memory operations wouldn't match, our simulations wouldn't capture that dependency. In addition, an 8-byte load might get its value from two 4-byte stores. This "partial forwarding" or "multiple forwarding" case is actually really nasty to get right (in simulation and even worse in real hardware), but it can actually happen in some codes, so real processors must handle it. We are ignoring it here.

As with the previous two experiments, run the code with the range of ROB sizes and plot the data on the same graph.

Experiment #4 and #5 - Realistic Branch Prediction

For the next experiment, we are going to consider realistic branch prediction. Extend your simulator to also model bimodal and gshare branch predictors.

Perform the branch prediction during fetch/rename. In a real machine, we would only know that the branch was mis-predicted once it executes (just like with memory addresses, above). However, in our trace-based simulations, we know immediately if the prediction was correct or not. This allows us to simplify the modeling of branch mis-predictions. Instead of actually flushing the pipeline, we can model a branch mis-prediction by stalling fetch until the branch resolves.

If the branch prediction was correct, continue as before. If the branch prediction is incorrect, fetch is suspended (no more instructions are fetched) and mark the branch in the ROB as "mispredicted" (you'll need to add another field to each entry in the ROB). At issue, check to see if the micro-op selected for execution is the mis-predicted branch micro-op. If so, set the "restart fetch counter" based on the execution latency plus the mis-prediction restart penalty. For these simulations, assume a restart penalty of five cycles (this represents the time it takes to redirect fetch, access the instruction cache, and start renaming instructions again.

Thus, adjust the fetch bases on the following code:

```
for (i=0; i<N; i++):
  if (FetchReady > 0):
    FetchReady = FetchReady - 1
    break;

  if (FetchReady == -1):
    break;

  assert(FetchReady == 0)

  if ROB.isFull():
    break;    // ROB is full
```

```
  else:
    Fetch/rename a micro-ops as described above
    ...
    ...
    if (fetched micro-op is a conditional branch):
      Make branch prediction
      if (fetched micro-op is a mis-predicted branch):
        FetchReady = -1;         // Suspend fetch
        microop.isMisPredictedBranch = true
```

During micro-op select, if a micro-op is selected to execute, also perform the following actions:

```
...
if (microop.isMisPredictedBranch):
  assert(FetchReady == -1)
  FetchReady = latency + restart_penalty;  // The mis-prediction restart penalty
...
```

> **Note**
>
> Let me reemphasize that in a real machine, the hardware wouldn't know the branch was mis-predicted, so it would go on to fetch instructions down the "wrong path", inject them into the instruction queue and execute them speculatively. As we are doing this based on a trace, we know immediately that we mis-predicted the branch, thus we can *approximate* the performance of speculative execution without actually implementing recovery and such. In fact, fetching down wrong paths is actually really difficult to model using traces, as we only have the "correct" trace of instructions. This is a limitation of simple trace-based simulations. In contrast, a simulation that actually simulates the execution of the program can actually model the performance (and energy) impact of executing down the wrong path.

To calculate the impact of branch prediction, perform simulations with both a bimodal and a gshare predictor. Both predictor use a table of 2^{16} two-bit counters (16KB of predictor, which is a reasonably large predictor). The gshare predictor uses 16 bit of history; you may wish to model the bimodal predictor as a gshare predictor with zero history bits.

In the set of debug traces, those labeled "experiment 4" are with the bimodal predictor and those labeled "experiment 5" are for the gshare predictor.

For the same range of ROB sizes, simulate the pipeline using three predictors: perfect (the data you already have from experiment #3), the bimodal predictor (experiment #4), and the gshare predictor (experiment #5). Plot these results as three lines on a *new* graph with the same x-axis and y-axis as previously. Label each line appropriately.

Discussion Questions

1. **For experiment #1** (instruction-level parallelism):
    a. What is the maximum achievable IPC?
    b. How large must the ROB be to achieve 4 IPC?
2. **For experiment #1/#2/#3** (memory scheduling):
    a. What is the maximum achievable IPC for conservative handling of memory scheduling?
    b. How large must the ROB be to achieve 4 IPC with conservative scheduling?
    c. For all the ROB sizes simulated, what is the largest difference between unrealistic and conservative memory scheduling? Give you answer in the form: unrealistic memory scheduling is x% faster than conservative memory scheduling.

   d. For all the ROB sizes you simulated, what is the largest difference between unrealistic and perfect memory scheduling? Give you answer in the form: unrealistic memory scheduling is x% faster than perfect memory scheduling.

3. **For experiment #4/5** (branch prediction):

   a. What is the maximum achievable IPC for the bimodal predictor configuration?

   b. What is the maximum achievable IPC for the gshare predictor configuration?

   c. What is the largest speedup provided by the gshare predictor over the bimodal predictor? Give your answer in the form: the pipeline with the gshare predictor is x% faster than the pipeline with the bimodal predictor.

What to Turn In

- **Short answers**. Turn in your answers to the above questions. I strongly urge you to type up your answers.
- **Graphs**. Turn in printouts of the following graphs. One per sheet of paper (to make them big enough to read). Label the axises and give each line a descriptive names in a legend.
  - Graph A: memory scheduling [Experiment #1 through #3]: The graph has the IPC versus ROB sizes for the first three experiments (one line per experiment). As stated above, the **x-axis is a log scale** to avoid most of the data points being squished into the left of the graph.
  - Graph B: branch prediction [Experiment #3 through #5]: The graph has the IPC versus ROB sizes for experiments 4 and 5 plus also includes the data from experiment 3 as a point of comparison (one line per experiment). As stated above, the **x-axis is a log scale** to avoid most of the data points being squished into the left of the graph.
- **Source Code**. Please print out your source code of the final simulator (not each intermediate state). The code should not be extremely long, but if your code is long you may wish to printing it out two-pages per sheet.

Addendum

- None, yet.
