Project #1 (Due February 18, 2014 – by mid-night)

NOTE: This assignment can be done by teams of two students – If you cannot find a partner send me an email. I will keep a list of students needing partners and form teams from this list.

The goal of this assignment is to simulate a 5 stage pipeline and an over-simplified superscalar and to compare different designs using real execution traces made available to you in trace files (file_name.tr). Each trace file is a sequence of trace items, where each trace item represents one instruction executed in the program that has been traced. A trace item is a structure:

```
      struct trace_item {
             uint8_t type;                  // see below
             uint8_t sReg_a;                // 1st operand
             uint8_t sReg_b;                // 2nd operand
             uint8_t dReg;                  // dest. operand
             uint32_t PC;                   // program counter
             uint32_t Addr;                 // mem. address
      };
```

where

```
      enum trace_item_type {
             ti_NOP = 0,

             ti_RTYPE,
             ti_ITYPE,
             ti_LOAD,
             ti_STORE,
             ti_BRANCH,
             ti_JTYPE,
             ti_SPECIAL,
             ti_JRTYPE
      };
```

The “PC” (program counter) field is the address of the instruction itself. The “type” of an instruction provides the key information about the instruction. Here is a more detailed explanation:

```
NOP - it's a no-op. No further information is provided.
RTYPE - An R-type instruction.
       sReg_a: first register operand (register name)
       sReg_b: second register operand (register name)
       dReg: destination register name
       PC: program counter of this instruction
       Addr: not used
ITYPE - An I-type instruction that is not LOAD, STORE, or BRANCH.
       sReg_a: first register operand (register name)
       sReg_b: not used
       dReg: destination register name
       PC: program counter of this instruction
       Addr: immediate value
LOAD - a load instruction (memory access)
       sReg_a: first register operand (register name)
       sReg_b: not used
       dReg: destination register name
       PC: program counter of this instruction
       Addr: memory address
STORE - a store instruction (memory access)
       sReg_a: first register operand (register name)
       sReg_b: second register operand (register name)
       dReg: not used
       PC: program counter of this instruction
       Addr: memory address
BRANCH - a branch instruction
       sReg_a: first register operand (register name)
       sReg_b: second register operand (register name)
       dReg: not used
       PC: program counter of this instruction
       Addr: target address
JTYPE - a jump instruction
       sReg_a: not used
       sReg_b: not used
       dReg: not used
       PC: program counter of this instruction
       Addr: target address
SPECIAL - it's a special system call instruction
       For now, ignore other fields of this instruction.
JRTYPE - a jump register instruction (used for "return" in functions)
       sReg_a: source register (that keeps the target address)
       sReg_b: not used
       dReg: not used
       PC: program counter of this instruction
       Addr: target address
```

To avoid dealing with binary files, you are given a program *pipeline.c* which reads a trace file (a binary file) and simulates a single cycle CPU (a very simple simulation). It outputs the total number of cycles needed to execute the instructions in the trace file, and if the *trace_view_on* switch is set, outputs also the details of the instruction that finished execution in each cycle. Hence, the program *pipeline.c*, which include *trace_item.h*, takes two arguments; the name of the trace file and a switch value (0 or 1). For example, when you execute "`pipeline sample.tr 1`" you get an output that looks like (if the second argument is 0 rather than 1, only the last line is printed):

```
[cycle 1] LOAD: (PC: 2000a0)(sReg_a: 29)(dReg: 16)(addr: 7fff8000)
[cycle 2] ITYPE: (PC: 2000a4)(sReg_a: 255)(dReg: 28)(addr: 1001)
[cycle 3] ITYPE: (PC: 2000a8)(sReg_a: 28)(dReg: 28)(addr: ffffc000)
[cycle 4] ITYPE: (PC: 2000ac)(sReg_a: 29)(dReg: 17)(addr: 4)
[cycle 5] ITYPE: (PC: 2000b0)(sReg_a: 17)(dReg: 3)(addr: 4)
[cycle 6] ITYPE: (PC: 2000b4)(sReg_a: 255)(dReg: 2)(addr: 2)
[cycle 7] RTYPE: (PC: 2000b8)(sReg_a: 3)(sReg_b: 2)(dReg: 3)
[cycle 8] RTYPE: (PC: 2000bc)(sReg_a: 0)(sReg_b: 3)(dReg: 18)
[cycle 9] STORE: (PC: 2000c0)(sReg_a: 28)(sReg_b: 18)(addr: 10004884)
[cycle 10] ITYPE: (PC: 2000c4)(sReg_a: 29)(dReg: 29)(addr: ffffffe8)
[cycle 11] RTYPE: (PC: 2000c8)(sReg_a: 0)(sReg_b: 16)(dReg: 4)
. . .
[cycle 21] STORE: (PC: 20cd7c)(sReg_a: 29)(sReg_b: 31)(addr: 7fff7fcc)
[cycle 22] BRANCH: (PC: 20cd80)(sReg_a: 16)(sReg_b: 0)(addr: 20cda8)
[cycle 23] LOAD: (PC: 20cd84)(sReg_a: 16)(dReg: 4)(addr: 7fff8007)
. . .
[cycle 32] BRANCH: (PC: 20a9a4)(sReg_a: 17)(sReg_b: 0)(addr: 20a9b4)
[cycle 33] RTYPE: (PC: 20a9b4)(sReg_a: 0)(sReg_b: 0)(dReg: 16)
....

 + Simulation terminates at cycle : 1000
```

PART 1 of this project is to replace the simple single cycle simulation with a simulation of the 5 stage pipeline (IF, ID, EX, MEM and WB) which will also output the total number of execution cycles as well as the instruction that exits the pipeline in each cycle (if the switch *trace_view_on* is set to 1). The simulated pipeline will assume that the architecture includes the forwarding hardware described in the book (from X/M register to ALU input and from M/W register to ALU input) and that the register file is updated in the first half of the WB stage and is read in the second half of the ID stage. Hence, the pipeline should stall only when a load instruction is followed by an instruction that uses the loaded data (a load-use hazard).

In addition to stalling due to data hazards, the architecture should assume that the branch (as well as the jump) target address and the branch condition are resolved in the EX stage. Consequently, your simulation should take a third argument, *prediction_method* (in addition to the trace file name and *trace_view_on*). This argument will be 0, or 1 to reflect two possible designs as follows:

- If *prediction_method = 0,* your simulation should assume that branches are always predicted as "not taken". In this case, if the prediction is wrong, the two instructions that entered the pipeline before the branch condition is resolved should be squashed.
- If *prediction_method = 1,* your simulation should assume that the architecture uses a one-bit branch predictor which records the last branch condition and address. This predictor is consulted in the IF stage which means that if the prediction is correct, the instruction following the branch in the pipeline is the correct one. Otherwise, the wrong prediction will be discovered when the branch is in the EX stage and the two instructions following the branch in the pipeline will be squashed. When no prediction can be made, the "predict not taken" policy should be assumed. Use a Branch Prediction Hash Table with 128 entries and index this table with bits 10-4 of the branch instruction address (note that some addresses will collide and thus some stored information will be lost – that is OK).

Your implementation should simulate the five pipeline stages and determine the instruction (or an inserted no-op to reflect stalling or squashing) that is in each stage at every cycle. Note that the traces you are given are dynamic traces, hence they do not show which instructions are following a wrong branch (the squashed instructions). You should introduce these squashed instructions into the pipeline without knowing what instructions they were exactly (they were squashed anyways – you should print them as SQUASHED in the output).

PART 2 of the project is to extend your simulation to a "very" simple version of a dynamically scheduled superscaler with two 5-stage pipelines, one for ALU and branch instructions and the other for load/store instructions. The first stage, called IF+ID fetches and decodes up to two instructions every cycle and stores them in an "*instruction buffer*". A dynamic scheduler will send up to two instructions every cycle from the IF+ID stage to the second stage, called REG, where the registers needed by the instructions are read. When an instruction is moved from IF+ID to REG, we say that the instruction is *issued*. The remaining three stages, EX, MEM and WB are specific to each of the two pipelines with all the needed forwarding paths and the associated control logic implemented within each pipeline and between the two pipelines. As in the single pipeline case, branches (and jumps) are resolved in the EX stage. Hence, if a branch is mis-predicted, the instructions that entered the pipeline in the two cycles following the branch should be squashed.

In every cycle, the dynamic scheduler looks at the two instructions in the instruction buffer and issues instructions to the REG stage according to the following rules:

1. If one of the instructions is a lw/sw instruction and the other is an ALU/branch instruction, the first of the two instructions is not a branch instruction, there is no data dependence among

the two instructions and neither of the two instructions have load/use dependence on a load instruction that was issued in the previous cycle, then both instructions are issued in the current cycle. Note that to implement this rule, the IF+ID stage needs to determine if a load instruction was issued in the previous cycle and keep track of the register that was loaded, if any.

2. Else, if the first of the two instructions does not have a load/use dependence on a load instruction that was issued in the previous cycle, then that instruction is issued to the appropriate pipeline and a no-op is issued to the other pipeline.
3. Else two no-ops are issued to the two pipelines (no actual instruction is issued in that cycle)

Note that the instruction buffer can hold only two instructions. Hence, if only one instruction is issued in a given cycle, then the IF+ID stage discards the second of the two instruction that it fetched during that cycle (will be fetched again in the next cycle). If no instructions are issued, then IF+ID discards the two instructions that it fetched during that cycle. Discarded instructions can be held in an overflow buffer to avoid fetching them again the next cycle.

TRACES: You are provided with 4 short and 3 long trace files (sample1.tr, sample2.tr, sample3.tr, sample4.tr) and (sample_large1.tr, sample_large2.tr, sample_large3.tr). These files are accessible at **/afs/cs.pitt.edu/courses/1541/long_traces** and **/afs/cs.pitt.edu/courses/1541/short_traces**, where you can also find *sample.tr*, a small trace file you can use while you debug.

Your assignment is to modify pipeline.c to simulate the 5 stage pipeline and to write a new simulator for the superscalar pipeline (superscalar.c). You should test your new pipeline.c and superscalar.c on the traces provided. You should submit the output of your simulation for each short and long trace file with trace_view_on = 0 and prediction_method = 0 and 1.

What to submit (email to the TA):

1) Your source codes for *pipeline.c* and superscalar.c that take 3 arguments; the input trace file, the branch *prediction_method* and the *trace_view_on* switch (in that order). The argument *prediction_method* should be 0 (for "predict not taken") or 1 (for 1-bit prediction). In case the last two parameters are not specified, their default values should be *prediction_method* = 0 and *trace_view_on* = 0.

2) The result of running your simulations with *trace_view_on* = 0 for each short and long trace file and *prediction_method* = 0, 1, and 2. Put the results in a table.

3) A short analysis of the effect of having two pipelines and of having a branch predictor (observed from the results).