Project #1 (Due February 19, 2018 – by mid night)

NOTE: This assignment can be done by teams of two or three students – If you cannot form a team, send me an email. I will keep a list of "team-less" students and form teams from this list.

One way to increase the clock frequency in the 5-stage pipeline is to split each of the IF and MEM stages into two stages each, and to allow either writing or reading of the register file in one cycle. The resulting pipeline will thus have 7 stages (IF1, IF2, ID, EX, MEM1, MEM2, WB) with a structural hazard resulting when an instruction in the WB stage writes into the register file while and instruction in the ID stage reads from the register file. A branch predictor predicts the outcome of a branch in the IF1 stage. The branch target address and condition are computed in the EX stage. The targets of jump instructions are also resolved in the EX stage and are included in the branch target buffer. Forwarding paths (and logic) are provided from the EX/MEM1 and the MEM2/WB buffers to the ID/EX buffer.

Hazards in the 7-stage pipeline can be avoided as follows:

1) Structural hazards: if the instruction at WB is trying to write into the register file while the instruction at ID is trying to read form the register file, priority is given to the instruction at WB. The instructions at IF1, IF2 and ID are stalled for one cycle while the instruction at WB is using the register file.

2) Data hazards:
    a. If an instruction in EX/MEM1 is a *load* instruction which will write into a register R while the instruction in ID/EX is reading from register R, then the instruction in ID/EX (and subsequent instructions) should stall since it does not have the correct content of register R. A no-op is injected into EX/MEM1.
    b. If an instruction in MEM1/MEM2 is a *load* instruction which will write into a register R while the instruction in ID/EX is reading from register R, then the instruction in ID/EX (and subsequent instructions) should stall to allow forwarding of the result from MEM2/WB to ID/EX (in the following cycle). A no-op is injected in MEM1/MEM2.

3) Control hazards: If, when a branch or jump is resolved in the EX stage, a control hazard is detected, the instructions in the IF1, IF2 and ID stages are flushed.

The goal of this assignment is to simulate the above 7-stages pipeline and experiment with different pipelined 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), sometimes not used (ex lui)
       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 *CPU.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 *CPU.c*, which includes *CPU.h*, takes two arguments; the name of the trace file and a switch value (0 or 1). For example, when you execute "`CPU 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: fffffe8)
[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
```

Note: When the register number is 255, this means that the instruction does not use this register. For example, the ITYPE instruction "*lui rt, imm*" loads an immediate constant into the destination register. It does not use a source register.

The project is to replace the simple single cycle simulation with a simulation of the 8-stages pipeline 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).

In addition to stalling/flushing due to hazards, 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, 1, or 2 to reflect three 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 three instructions that entered the pipeline before the branch condition is resolved should be flushed.
- 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 IF1 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 in the EX stage and the three instructions following the branch in the pipeline will be flushed. When no prediction can be made, the "predict not taken" policy should be assumed. Use a Branch Prediction Hash Table with 64 entries and index this table with bits 8-3 of the branch instruction address (note that some addresses will collide and thus some stored information will be lost – that is OK).

- If *prediction_method = 2*, your simulation should assume that the architecture uses a 2-bit branch predictor.

Your implementation should simulate the 7 pipeline stages and determine the instruction (or an inserted no-op as a result of stalling or flushing) 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).

TRACES: You are provided with 4 short and 2 long trace files (sample1.tr, sample2.tr, sample3.tr, sample4.tr) and (sample_large1.tr, sample_large2.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 CPU.c to simulate the 7-stages pipeline and test your simulation 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, 1, and 2.

What to submit (email to the TA):

1) Your source code for a *CPU.c* simulator which takes 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"), 1 (for "1-bit branch predictor"), or 2 (for 2-bit predictor). 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 simulation 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 and write down a short analysis of the effect of the branch predictors (observed from the results).

3) Change the size of the prediction table (should be a parameter in your program) to 32 and 128 (instead of 64). Compare the results for the three different table sizes and comment on the effect of the size of the prediction table.

NOTES:
- Prior to submission, make sure your code compiles and runs on the CSSD Unixs cluster, which will be used for grading. All submitted results must be generated on the Unixs cluster.
- The order of command line arguments for your simulator must be:
  input trace file, prediction_method, trace_view_on
- In addition to the source code, you should submit a single pdf with all the necessary results and analysis.

Debugging help:

The trace generator (trace_generator.c) may be used to manually build your own trace files. This program takes the name of the trace file that you want to create as a command line argument and will prompt you for the 5 fields of "trace_item" for each instruction that you want to include in the trace. You may test the correctness of your simulation by testing it on multiple short traces that you specifically create to test specific features/scenarios. In addition to testing your program on the given traces, we will also test it on some short traces that we will design to check correctness.