Project #1 (Due February 18, 2019 – by mid night)

The goal of this project is to compare 5-stage and 8-stage pipeline architectures and assess the effect of branch prediction through simulation. It should be done by teams of three students.

Project Setup:

The goal of this project is to simulate pipeline architectures and experiment with different designs using real execution traces made available to you in trace files (file_name.tr). You will experiment with 4 short trace files (sample1.tr, sample2.tr, sample3.tr, sample4.tr) accessible at **/afs/cs.pitt.edu/courses/1541/short_traces** and 2 long trace files (sample_large1.tr, sample_large2.tr) accessible at **/afs/cs.pitt.edu/courses/1541/long_traces**. In the first directory, you will also find *sample.tr*, a small trace file you can use while you debug. Each trace file is a sequence of trace items, where each item represents one instruction executed in the program that has been traced. A trace item is stored in a structure:

```
struct instruction {
        uint8_t type;               // holds the op-code. 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 the type is an enumeration of possible op-codes. Specifically,

```
enum opcode{
        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. A detailed list of instructions is given below:

```
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 *five_stage.c* which reads a trace file (a binary file containing a sequence of instructions) and simulates a bare-bones 5-stage pipeline. It outputs the total number of cycles needed to execute the instructions in the trace file. It also outputs the details of the instruction that finished execution in each cycle if a specific switch, *trace_view_on*, is set.

First, you should compile the program *five_stage.c*, which includes *CPU.h*. It takes two arguments; the name of the trace file and a switch value (0 or 1). Make sure that when you execute "*five_stage sample.tr 1*" you get the following output (if the second argument is 0, only the last line is printed):

[cycle 5] LOAD: (PC: 2097312)(sReg_a: 29)(dReg: 16)(addr: 2147450880)
[cycle 6] ITYPE: (PC: 2097316)(sReg_a: 255)(dReg: 28)(addr: 4097)
[cycle 7] ITYPE: (PC: 2097320)(sReg_a: 28)(dReg: 28)(addr: -16384)
[cycle 8] ITYPE: (PC: 2097324)(sReg_a: 29)(dReg: 17)(addr: 4)
[cycle 9] ITYPE: (PC: 2097328)(sReg_a: 17)(dReg: 3)(addr: 4)
[cycle 10] ITYPE: (PC: 2097332)(sReg_a: 255)(dReg: 2)(addr: 2)
[cycle 11] RTYPE: (PC: 2097336)(sReg_a: 3)(sReg_b: 2)(dReg: 3)
[cycle 12] RTYPE: (PC: 2097340)(sReg_a: 0)(sReg_b: 3)(dReg: 18)
[cycle 13] STORE: (PC: 2097344)(sReg_a: 28)(sReg_b: 18)(addr: 268454020)
[cycle 14] ITYPE: (PC: 2097348)(sReg_a: 29)(dReg: 29)(addr: -24)
[cycle 15] RTYPE: (PC: 2097352)(sReg_a: 0)(sReg_b: 16)(dReg: 4)
....
[cycle 26] BRANCH: (PC: 2149760)(sReg_a: 16)(sReg_b: 0)(addr: 2149800)
[cycle 27] LOAD: (PC: 2149764)(sReg_a: 16)(dReg: 4)(addr: 2147450887)
…

```
[cycle 36] BRANCH: (PC: 2140580)(sReg_a: 17)(sReg_b: 0)(addr: 2140596)
[cycle 37] RTYPE: (PC: 2140596)(sReg_a: 0)(sReg_b: 0)(dReg: 16)
....
+ Simulation terminates at cycle : 1004
```

Note: when the register number is not between 0 and 31, 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.

Your assignment is to modify *five_stage.c* to deal with hazards in the 5 stage pipeline. You will also write a new simulator for a pipeline with 8 stages (*eight_stage.c*). In both simulations, you will assume that the architectures have forwarding/stalling hardware and logic and that branch targets and conditions are computed in the ID stage (targets of jump instructions are also computed in the ID stage). You will test your new programs on the traces provided and submit your code and the results of simulating different designs.

Milestone 1 (Modifying five_stage.c to avoid hazards and add branch prediction):

Assuming that the hardware implements forwarding and stalling, you should enhance the simulation of the 5-stage pipeline with hazard detection which will stall the pipeline (and insert a no-op) when a load instruction is in the EX stage while the instruction in the ID stage uses the loaded data (a load-use hazard).

In addition to stalling due to data hazards, you should also deal with control hazards. Specifically, you should simulate the case when the instruction following a taken branch enters the pipeline before the condition of the branch is resolved (such instruction will be squashed/flushed). Note that the traces you are given are dynamic traces that list the sequence of executed instructions when the program is executed on a single cycle machine which does not have any hazards. Hence, from the traces you cannot find out which instructions syntactically follow a taken branch. However, for any branch instruction in the trace, you can find out if it is taken by comparing the "target address field" of the branch instruction with the PC of the next instruction in the trace. If the "target address field" is equal to the PC of the next instruction, then this indicates that the branch was taken (as is the case in the branch instruction shown in cycle 36 in the above simulation output). Otherwise, the PC of the next instruction in the trace is equal to the PC of the branch instruction plus 4, which indicates that the branch was not taken (as is the case in the branch instruction shown in cycle 26). When you discover that a branch is taken, you can simulate the instruction that enters the pipeline after the branch without knowing its type by denoting it as "FLUSHED".

Finally, you should add branch prediction. Specifically, your simulator 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 there is no branch predictor, which is equivalent to always predicting that a branch is "not taken".

- If prediction_method = 1, your simulation should assume that the architecture uses a one-bit branch/jump predictor which records the condition and target of the last execution of a branch as well as the target of the last execution of a jump. This predictor is consulted in the IF stage. If the prediction turns out to be incorrect, the instruction(s) following the branch in the pipeline should be squashed/flushed. Use a Branch Prediction Hash Table with 64 entries and index this table with bits 9-4 of the branch instruction address (note that some addresses will collide and thus some stored information will be lost – that is OK).

Milestone 2 (Simulating the 8-stage pipeline):

One way to increase the clock frequency in the 5-stage pipeline is to split each of the IF, EX 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 8 stages (IF1, IF2, ID, EX1, EX2, 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 can predict the outcome of a branch in the IF1 stage.

Structural hazards in the 8-stage pipeline can be avoided as follows: 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.

To efficiently deal with data hazards, forwarding paths are provided from the EX2/MEM1, MEM1/MEM2 and the MEM2/WB buffers to the ID/EX1 buffer. Note that there is no forwarding path between EX1/EX2 and ID/EX1 because the result of the ALU is only available at the end of the EX2 stage. Given the forwarding paths, data hazards can be avoided as follows:

a. If at the beginning of a given cycle, the instruction in EX1/EX2 will write into a register R while the instruction in ID/EX1 already read from register R, then the instruction in ID/EX1 (and subsequent instructions) should stall since the correct content of register R is not yet available for forwarding. A no-op should be injected into EX1/EX2 at the end of the cycle.
b. If at the beginning of a given cycle, the instruction in EX2/MEM1 or the instruction in MEM1/MEM2 is a *load* instruction which will write into a register R while the instruction in ID/EX1 already read from register R, then the instruction in ID/EX1 (and subsequent instructions) should stall since the correct content of register R is not yet available for forwarding. A no-op should be injected into EX1/EX2.

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

What your team should submit to the TA via email:

1) Your source codes for the *five_stage.c* and *eight_stage.c* simulators. Each simulator should take 3 arguments; the name of the input trace file, the branch *prediction_method* and the *trace_view_on* switch (in that order). Grading will be done through a script, so you should not change the format of the output produced when *trace_view_on* =1.

2) The result of running your simulations with *trace_view_on* = 0 for each short and long trace file and *prediction_method* = 0 and 1. Put the results in a table and write down a short analysis of the effect of increasing the number of pipeline stages as well as the effect of the branch predictors on each pipeline (observed from the results). Make a recommendation about which of the 5-stage or 8-stage pipeline is more efficient assuming that the clock frequency of the 8-stage pipeline is double that of the 5-stage pipeline.

3) Experimenting only using the 5-stage pipeline, change the size of the prediction table (should be a parameter in your program) to 32 and 128 (using bits 9-5 and 10-4 of the PC, respectively). Compare the results for the three different table sizes (32, 64 and 128) and comment on the effect of the size of the prediction table.

Grading Criteria:

Your project will be graded according to the following criteria:

- 25 points for meeting M1 and passing tests that show that you correctly handle the potential hazards
- 15 points for meeting M2 and passing tests that show that you correctly handle the potential hazards
- 20 points for providing experimental comparisons and analysis of the results.

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 prompts you for the 5 fields of "trace_item" for each instruction that you want to include in the trace. You may check the correctness of your simulation by testing it on multiple short traces that you specifically create to test specific features/scenarios. This is the same trace generator that the TA will use to generate test traces for checking the correctness of the simulators.

Notes:

- Prior to submission, make sure your code compiles and runs and all your results are generated on the CS Linux cluster, which is accessible through "ssh **linux.cs.pitt.edu**". Your project will only be tested on this 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. All submissions of written content should be in a single pdf file. Multiple result files or written submissions will not be accepted.