# CSCI 2500 Group Project

April 7, 2017

## Instruction Pipeline and Cache (IPLC) Simulator

### Due Date: Wednesday, May 3rd, 2017

### Overview

For this GROUP assignment (up to 4 students per team) you will be implementing and comparing various implementations of a combined instruction pipeline and cache (IPLC) simulator. In particular, your group will be determining what design yields the best overall performance (i.e., fewest cycles). For the pipeline/-cache simulation portion, you will be given a template which you will need to extend including the following set of files:

- **instruction-trace.txt:** This is an instruction trace file. The format is described below.
- **iplc-sim.c:** template for the simulator code.
- **Makefile:** builds the iplc simulator.
- **taken-2-2-2.out:** trace-file output for the case described below. Used to help you debug your implementation.

### Instruction Trace Format

Figure 1, provides a example of the nearly 35,000 plus MIPS instruction trace. This trace is from a 2-D Matrix Swap and Multiply MIPS code.

The format for each instruction is as follows. First, each instruction occupies a single line of text. The first field is the memory address at which the instruction is located. Next, follows the MIPS instruction mnemonic (e.g., `lw` for Load Word followed by the registers and or offset that might be used as part of the instruction. A colon (:) is used to denote the instruction has ended and the address that follows is the absolute memory DATA address that was read or written by that instruction if it accessed memory. Instructions that do not access memory will not have a colon nor DATA address following.

**NOTE: I have provided you with the code necessary to parse this instruction trace file.**

### Cache Details

Using this cache simulator and INSTRUCTION and DATA address trace (which the cache simulator skeleton code reads in for you), your overall goal is to find the optimal values for (1) block size, and (2) associativity such that the miss rate of the cache is minimal, yet the total size of the cache must **be less than 10240 bits**. Notice the total size of the cache is expressed in *bits* and not *bytes*!. Possible cache configurations are the following:

```
0x00400000  lw $4, 0($29): 7fffef48
0x00400004  addiu $5, $29, 4
0x00400008  addiu $6, $5, 4
0x0040000c  sll $2, $4, 2
0x00400010  addu $6, $6, $2
0x00400014  jal 0x00400024
0x00400024  addi $29, $29, -4
0x00400028  sw $31, 0($29): 7fffef44
0x0040002c  lui $4, 4097
0x00400030  lui $1, 4097
0x00400034  ori $5, $1, 200
0x00400038  ori $6, $0, 5
0x0040003c  jal 0x0040025c
0x0040025c  ori $8, $0, 4
0x00400260  add $9, $0, $0
0x00400264  beq $9, $8, 180
0x00400268  ori $10, $0, 2
0x0040026c  beq $10, $8, 164
0x00400270  add $11, $9, $9
0x00400274  add $11, $11, $11
0x00400278  add $11, $11, $11
0x0040027c  add $11, $11, $11
0x00400280  add $12, $10, $10
0x00400284  add $12, $12, $12
0x00400288  add $11, $11, $12
0x0040028c  add $11, $11, $4
0x00400290  lw $13, 0($11): 10010008
```

Figure 1: Example instruction trace from the MIPS 2-D Matrix Swap and Multiple Program. There are 34,753 instructions in this trace!

- **Possible Block Sizes:**
  - 1 word (32 bits)
  - 2 words (64 bits)
  - 4 words (128 bits)
- **Levels of Associativity**
  - direct mapped (1-way set associative)
  - 2-way set associative
  - 4-way set associative

**Cache Size Function:** You can express the size of the cache in terms of 3 factors:

- BlockSize
- Associativity
- IndexBits

$CacheSize = Associativity * (2^{IndexBits} * (32 * BlockSize + 33 - IndexBits - BlockOffSetBits))$

where $BlockOffSetBits$ is the $Log_2$ of the BlockSize.

Using the above equation (which has been coded for you in the cache skeleton), you need to find the largest value for the index such that it results in a total cache size that is less than or equal to maximum cache size

of 10,240 (10k) bits FOR EACH combination of blocks size and level of associativity. Thus, there will be 9 value combinations of index size, block size and level of associativity. Your cache simulator should compute this.

Use your software skills to build a "cache" data structure that is dynamic (i.e., use `malloc()`). Meaning, I do not want to see 9 hard coded cache data structures that correspond to the above 9 different configurations.

You are to assume that on reads and writes, the data is brought into the cache – i.e., to simplify things, you will have a "write/miss" in this cache.

In order for you to check yourself, I have provided the pipeline trace and cache performance output for the 2 bits of cache index, 2 word block size, 2-way set associative, branches taken case. Note, this case is not the largest cache size you could have had and is solely to help you debug.

As you will see, for this very small cache (only 2480 bits) the cache miss rate is high (i.e., $> 16\%$). You are allowed to have a MAX cache size of 10240 bits. So, you need to determine the 9 possible configurations using the template such that each configuration is the maximum index (number of cache lines) for a given block size and level of associativity. Equally, if your cache simulator does not get this precise answer, then your cache simulator is incorrect and you will not get the performance in cycles correct.

## Pipeline Details

The pipeline is the standard 5-stage design consisting of FETCH, DECODE, ALU, MEM and WRITEBACK stages. The design features you are to consider are:

- On a cache instruction MISS, the pipeline FETCH stage incurs a **stall penalty** of 10 clock cycles.
- On a data word MISS (LW/SW instruction) is also a 10 clock cycle **stall penalty** in the MEM stage for LW and WRITEBACK stage for SW.
- Determine which branch prediction method yields better performance, TAKEN or NOT-TAKEN?
- Last, you are to "forward" registers across stages to avoid pipeline stalls.

Note, you will know if a branch is taken because you can see if the address for the next instruction is 4 bytes larger or not. That is, if $PC + 4$ was not the next instruction, then you know the branch was taken. Thus, if your predictor guesses wrong, you just need to insert the NOP instruction into the pipeline but do not count that as a real instruction but count the cycles.

The forwarding logic can be found in the textbook. Note, however, your logic is much easier. You will know what instruction you are processing as opposed to particular control signal lines. For example, if either `Reg1` or `Reg2` being used in the DECODE stage are the `Dest_Reg` in the ALU, or MEM stages, you can FORWARD either `Reg1` or `Reg2` to avoid the stall. That is, when you detect this condition, you'll allow the pipeline to continue without stalling (e.g., adding delay cycles to overall cycle count). Also note that if a `Dest_Reg` in the WRITEBACK stage is used in the DECODE stage, then no forwarding is needed since the write occurs in the first half of the clock cycle and the read occurs in the last half of the clock cycle for that pipeline stage.

The functions you will write for this functionality are detailed in comments provided in the template code `iplc-sim.c` which I will cover in class.

## Performance Evaluation

You will run your simulator 18 times – 9 cache configurations plus branch predictor configured TAKEN and 9 cache configurations plus branch predictor configured NOT-TAKEN.

**Show the summary output for all your runs and indicate which configuration performed best for this instruction trace file.**

## Grading Criteria

The following is how these projects will be graded.

1. The project is viewed as 3 major parts: Cache Simulator (20 points), Pipeline Simulator (35 points), and Performance Evaluation (20 points). You should provide good comments in your code (15 points) so we can follow what you are doing – this is especially true if something does not work right. You can at least explain what you where trying to accomplish with your comments.
2. Write-up and documentation (detailed below) is worth 10 points.
3. If you have a logic error with your cache or pipeline simulator, you will only be deducted points from that part. So, suppose your cache simulator does not work well at all, then you could obtain a score of 80 points if all other parts are done well.
4. If your simulator seg faults or dumps core due to an error, the most points you can obtain is 25. Having code written that does not work correctly hurts you much more than having much less code that actually does useful operations.

## Report Write-up Instructions

Provide a write-up that summarizes your implementation. Describe how each team member contributed to the overall project. Please note, that while not all team members need to be involved in the final project write-up process, it is expected that ALL team members participate and contribute to the design and implementation process. This part of the project document should come first.