Lab 3: Simulating Caches and Branch Prediction

Assigned: Mon., 13.11; Due: **Mon., 27.11** (Midnight)

Instructor: Onur Mutlu
TAs: Hasan Hassan, Arash Tavakkol, Mohammad Sadr, Lois Orosa, Juan Gomez Luna

1. Introduction

In this lab, you will implement a *timing simulator* (written in C) to model instruction/data caches and a branch predictor. Unlike the RTL that you have developed in previous labs, a *timing simulator* is *not* a direct or synthesizable implementation of the processor. Rather, it is a higher-level, abstract model designed to allow quick architectural exploration. Describing and simulating hardware at a higher level of abstraction allows the designer to quickly see how design choices (e.g., caches or branch predictors) would impact performance.

We will give you the base simulator (it has been developed to match the processor that we have used in previous labs). We will also fully specify the behavior of the caches and the branch predictor. Your job is to extend the simulator so that it implements the caches and the branch predictor as specified.

2. Timing Simulator

Unlike in previous labs, where we worked with a pipeline in Verilog, we are not constrained to logic-level implementation in a C-based timing simulator. Our main goal is to compute the *number of cycles* a program execution would take on the simulated processor. Because of this, many simplifications are possible. We do not actually need to model control logic and datapath details in each block of the processor; we only need to write code for each stage that performs the relatively high-level function of that pipeline stage (read the register file, access memory, etc.). In general, the simulator's algorithms and structures *do not* need to *exactly* match the processor's algorithms and structures, as long as the *result* is the same. While we no longer have a low-level implementation, and thus cannot determine the critical path (or other cost metrics that we care about, such as area taken on a silicon chip, or power consumed during operation), we can know how many cycles a program would take (assuming we model the cycle-level solution accurately).

3. Your Task: Additions to the Baseline Timing Simulator

Your goal is to *implement* the timing simulator so that it models a MIPS machine with *(i)* instruction/data caches, and *(ii)* a branch predictor. In the following, we will fully specify the microarchitecture of the MIPS machine that you will simulate.

3.1. Instruction Cache

The *instruction cache* is accessed every cycle by the fetch stage.

Organization. It is a **four-way** set-associative cache that is **8 KB** in size with **32-byte** blocks (this implies that the cache has **64** sets). When accessing the cache, the set index is calculated using bits `[10:5]` of the PC.

Miss Timing. When the fetch stage *misses* in the instruction cache, the block must be retrieved from main memory. An access to main memory takes **50** cycles. On the 50th cycle, the new block is inserted into the cache. In total, an instruction cache miss stalls the pipeline for 50 cycles.

Replacement. When a new block is retrieved from main memory, it is inserted into the appropriate set within the instruction cache. If any way within the set is empty (i.e., invalid), the new block is simply inserted into the invalid way. However, if none of the ways in the set are empty, the new block *replaces* the *least-recently-used* block in the set. For both cases, the new block becomes the *most-recently-used* block.

Control-Flow. While the fetch stage is stalled due to a miss in the instruction cache, a control-flow instruction further down the pipeline may *redirect* the PC. As a result, the pending miss may turn out to be unnecessary: it is retrieving the wrong block from main memory. In this case, the pending miss is *canceled*: the block that is eventually returned by main memory is *not* inserted into the cache – even if the redirection happens on the very last stall cycle.[1] Finally, note that when the PC is redirected to an address pointing to the *same* block as the pending miss, the pending miss should *not* be cancelled.

3.2. Data Cache

The *data cache* is accessed whenever a load or store instruction is in the memory stage.

Organization. It is an **eight-way** set-associative cache that is **64 KB** in size with **32 byte** blocks (this implies that the cache has **256** sets). When accessing the cache, the set index is calculated using bits `[12:5]` of the data address that is being loaded/stored.

Miss Timing & Replacement. Miss timing and replacement of the data cache are identical to those of the instruction cache.

Handling Stores. Both load and store misses stall the pipeline for 50 cycles. They both retrieve a new block from main memory and insert it into the cache.

Dirty Evictions. When a "dirty" block is replaced by a new block from main memory, it must be written back into main memory. For the purpose of this lab, we will assume that such dirty evictions are handled *instantaneously* – i.e., they are written immediately into main memory in the same cycle as when the new block is inserted into the cache.

3.3. Assumptions about Instruction & Data Caches

- Assume that both caches are initially empty (i.e., all blocks are invalid).
- In both caches, every block has a separate tag that stores information about the block: e.g., address, valid, recency, etc. Tags are initialized to 0.
- Assume that the program that runs on the processor *never* modifies its own code (referred to as self-modifying code): a given block *cannot* reside in both the caches.

3.4. Branch Predictor

Organization. The *branch predictor* consists of *(i)* a *gshare* direction predictor and *(ii)* a *branch target buffer.*

Gshare Direction Predictor. The gshare predictor uses an **8-bit** global branch history register (GHR). The most recent branch is stored in the **least-significant-bit** of the GHR and a value of **'1'** denotes a taken branch. The predictor XORs the GHR with bits `[9:2]` of the PC and uses this 8-bit value to index into a **256-entry** pattern history table (PHT). Each entry of the PHT is a **2-bit** saturating counter that operates as discussed in class: a taken branch increments whereas a not-taken branch decrements; the four values of the counter correspond to strongly not-taken (00), weakly not-taken (01), weakly taken (10), strongly taken (11).

Branch Target Buffer. The branch target buffer (BTB) contains **1024 entries** indexed by bits `[11:2]` of the PC. Each entry of the BTB contains *(i)* an address tag, indicating bits `[31:12]` of the

[1] Note that this might not be a good idea, but this is our specification of the processor. If you are interested in the effects of "wrong-path" execution on processor performance, please see Mutlu et al. [1, 2].

PC; *(ii)* a valid bit; *(iii)* a bit indicating whether this branch is *unconditional*; and *(iv)* the target of the branch.

Prediction. At every fetch cycle, control logic in the predictor indexes into both the BTB and the PHT. If the predictor misses in the BTB (i.e., address tag != PC`[31:12]` or valid bit = 0), then the next PC is predicted as **PC+4**. If the predictor hits in the BTB, then the next PC is predicted as the **target** supplied by the BTB entry when either of the following two conditions are met: *(i)* the BTB entry indicates that the branch is unconditional, or *(ii)* the gshare predictor indicates that the branch should be taken. Otherwise, the next PC is predicted as **PC+4**.

Update. The GHR is updated immediately after the prediction without waiting for the branch to resolve.[2] The correct GHR should be restored upon a flush. The rest of branch predictor structures are updated in the execute stage, where all branches are resolved. This part of the update consists of: *(i)* updating the PHT, which is indexed using the value of the GHR that was used for prediction and *(ii)* updating the BTB. Unconditional branches *do not* update the PHT or the GHR, but they update the BTB (setting the *unconditional* bit in the corresponding entry).

Initial State. All branch predictor structures are initialized to 0.

3.5. Flushing the Pipeline

When resolving a branch, the pipeline is flushed under any of the following conditions:

- The instruction is a branch, but the predicted direction does not match the actual direction.
- The instruction is a branch, and it is taken, but the predicted destination (target) does not match the actual destination
- The instruction is a branch, but it was not recognized as a branch (i.e., BTB miss)
- You will need to ensure the correct value of GHR and the correct value of PC are restored upon a flush (i.e., a redirection of the fetch stage).

4. Lab Resources

4.1. Source Code

Do **NOT** modify any files or folders unless explicitly specified in the list below.

- `Makefile`
- `run`: Script that runs your simulator and compares it against the baseline simulator
- `src/`: Source code (**Modifiable; feel free to add more files**)
  - `pipe.c`: Your simulator (**Modifiable**)
  - `pipe.h`: Your simulator (**Modifiable**)
  - `mips.h`: MIPS related pound defines
  - `shell.c`: Interactive shell for your simulator
  - `shell.h`: Interactive shell for your simulator
- `inputs/`: Example test inputs for your simulator (**Modifiable; feel free to add more files**)

4.2. Makefile

We provide a `Makefile` that automates the compilation and verification of your simulator.

[2] For reference, please see Hao et al [3].

The first time you use the Makefile you should compile the baseline simulator:

```
$ make basesim
```

This will generate basesim, which is the baseline simulator corresponding to the code we provide. You can use it to verify the output of a program you run on your simulator. Note that the output of a program should always match the output obtained by running the program on the baseline simulator. However, the execution time of a program on the two simulators will not be same after your changes on the caches and the branch predictor.

To compile your simulator:

```
$ make
```

To compile your simulator and check it against the baseline simulator using one or more test inputs:

```
$ make run INPUT=inputs/inst/addiu.x
$ make run INPUT=inputs/inst/*.x
$ make run
```

5. Getting Started & Tips

5.1. The Goal

We provide you with a skeleton of the timing simulator that models a five-stage MIPS pipeline: pipe.c and pipe.h. As it is, the simulator is already architecturally correct: it can correctly execute any arbitrary MIPS program that only uses the implemented instructions.[3] When the simulator detects data dependences, it correctly handles them by stalling and/or bypassing. When the simulator detects control dependences, it correctly handles them by stalling the pipeline as necessary.

By executing the following command, you can see that your simulator (sim) does indeed have identical architectural outputs (e.g., register values) as the baseline simulator (basesim) for all the test inputs that we provide in inputs/.

```
$ make run
```

Your job is to model accurately the timing effects of the caches, the branch predictor, and the main memory in your timing simulator.

5.2. Studying the Timing Simulator

Please study pipe.c and pipe.h in detail.

The simulator models each pipeline stage as a separate function – e.g., pipe_stage_fetch(). The simulator models the state of the pipeline as a collection of pointers to Pipe_Op structures (defined in pipe.h). Each Pipe_Op represents one instruction in the pipeline by storing all of the necessary information about the instruction that is needed by the simulator. A Pipe_Op structure is allocated when an instruction is fetched. It then flows through the pipeline and eventually arrives at the last stage (writeback), where it is deallocated once the instruction completes. To elaborate, each stage receives a Pipe_Op from the previous stage, processes the Pipe_Op, and passes it down to the next stage. The simulator models pipeline stalls by stopping the flow of Pipe_Op structures and pipeline flushes by deallocating the Pipe_Op structures at different stages.

[3] This is not entirely true since we pose the usual restrictions on system calls, exceptions, etc.

5.3. Tips

- **Please do not distribute the provided program files. These are for exclusive individual use of each student of the Computer Architecture course. Distribution and sharing violates the copyright of the software provided to you.**
- **Read this handout in detail.**
- **If needed, please ask questions to the TAs using the online Q&A forum in Moodle.**
- When you encounter a technical problem, please first read the error messages. A search on the web can usually solve many debugging issues, and error messages.
- The two major tasks of this lab – implementing caches, and implementing branch prediction – are independent, so you may tackle them in either order. One way to approach this lab is to first write a generic implementation of a set-associative cache, and then plug it into both the fetch stage (instruction cache) and the memory stage (data cache). Once the caches are functional and they also stall the pipeline correctly, you can implement the branch predictor.

6. Submission

Use the corresponding assignment in Moodle (`https://moodle-app2.let.ethz.ch/`). You should submit all the files needed to compile and simulate your code in a single tarball (with the name `lab3_`*YourSurname*_*YourName*`.tar.gz`). Please include comments to explain what you have done in the simulator code.

7. Extra Credit

We will offer up to 50% additional credit *for this lab* for exploring two different design aspects of the cache, and another 50% additional credit for design aspects of the branch predictor.

7.1. Cache Exploration

1. **Cache size, block size, associativity:** a sweep of *cache parameters*. You should write a set of benchmarks that use significant amounts of memory (for example, accessing a large array in streaming or random patterns), and run your simulator to measure IPC for various cache parameters. Show how changing the associativity, block size, and cache size affect performance.
2. **Replacement and insertion policies:** an exploration of cache replacement and/or insertion policies. The *cache replacement policy* specifies which cache block in a set is replaced when a new block is inserted into the cache. The *cache insertion policy* specifies where in the list of blocks the new block is placed. Up to now, we have used a replacement policy that evicts (replaces) the least-recently-used block, and an insertion policy that places new blocks at the most-recently-used position. However, other replacement and insertion policies have been studied, and some have been shown to achieve significantly better performance (fewer cache misses) for certain access patterns [4, 5]. You should experiment with a variety of test programs and optimize the cache replacement/insertion policy.
3. **Other:** Optionally, you may also choose to experiment with other aspects of the cache. For example, using more sophisticated hashing functions to map cache blocks to cache sets and/or using more than one hashing function [6]. Implementing a victim cache is another possibility [7]. Since this part is open-ended, the instructor reserves the amount of extra credit that can be obtained but 35% extra credit (in addition to the 50% mentioned above) is possible, depending on the difficulty of the optimization and the goodness of the resulting design and implementation.

Please write a report (`report_cache.pdf`) that briefly summarizes *(i)* your observations on the effect of each cache parameter, *(ii)* your findings on cache replacement/insertion policies, and *(iii)* any other optimizations you implement. Your report does not need to be more than four pages, but feel free to use more pages to present schematics, data, and graphs. Please also submit the version of your simulator (`src/`) that implements the best performing cache designs. This version of the simulator and the report should be submitted as a separate folder (called *extra_credit_cache*) within the same tarball as the regular submission.

7.2. Branch Predictor Exploration

1. **Reducing interference in the PHT:** In the gshare predictor you implemented, the GHR is hashed with the branch PC in order to achieve better utilization of the PHT. Other designs with the same goal are:
   (a) the agree predictor [8]
   (b) the gskew predictor [9]

   You may choose to implement either one of them or both, and compare them to your baseline gshare predictor.
2. **Other branch predictors:** Implement other branch predictors to improve the branch prediction accuracy of your processor and compare it to your baseline gshare predictor. We will hold a competition on branch prediction accuracy. Among all submissions, the top students that have the best branch prediction accuracy and the best branch predictor designs will receive up to 75% additional credit *for this lab* as well as "prizes" (at the discretion of the instructor). You may choose any or all of the following:
   (a) Perceptron branch predictor [10]
   (b) TAGE branch predictor [11]
   (c) Tournament branch predictor [12, 13]
   (d) Any other branch prediction mechanism, including optimization of the gshare predictor. You can find more branch prediction proposals in `https://www.jilp.org/cbp2016/program.html`.
3. **Other:** Optionally, you can explore other control-flow handling mechanisms. For instance, you may want to support predication [14, 15] in your pipeline. Implementing the conditional move instruction (`CMOV`) in your machine might be a simple way to support predication. You will also have to manually modify some programs to remove branches and include conditions evaluation. Since this part is open-ended, the instructor reserves the amount of extra credit that can be obtained but 35% extra credit is possible (in addition to the 50% mentioned above), depending on the difficulty of the mechanism and the goodness of the resulting design and implementation.

Please write a clear and detailed-enough report (`report_branch.pdf`) that summarizes your findings for *(i)* the effectiveness of the different designs to reduce PHT interference, *(ii)* the comparison between the baseline gshare predictor and the other branch predictors, and *(iii)* any other optimizations you choose to implement and explore. Your report does not need to be more than four pages, but feel free to use more pages for schematics, data, and graphs. Please also submit the versions of your simulator (as many `src/` folders as needed) that implement the different predictors. These versions of the simulator and the report should be submitted as a separate folder (called *extra_credit_branch*) within the same tarball as the regular submission.

References

[1] O. Mutlu et al. Understanding the effects of wrong-path memory references on processor performance. In *WMPI*, 2004.

[2] O. Mutlu et al. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors. In *IEEE TC*, 2005.

[3] E. Hao et al. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *MICRO*, 1994.

[4] M.K. Qureshi et al. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[5] V. Seshadri et al. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *PACT*, 2012.

[6] A. Seznec. A case for two-way skewed-associative cache. In *ISCA*, 1993.

[7] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA*, 1990.

[8] E. Sprangle et al. The agree predictor: A mechanism for reducing negative branch history interference. In *ISCA*, 1997.

[9] A. Seznec. An optimized 2bcgskew branch predictor. In *IRISA Tech. Report*, 1993.

[10] D.A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA*, 2001.

[11] A. Seznec. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction Level Parallelism*, 2006.

[12] R.E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 1999.

[13] S. McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.

[14] J.R. Allen et al. Conversion of control dependence to data dependence. In *POPL*, 1983.

[15] H. Kim et al. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *MICRO*, 2005.