built on 2020/10/16 at 18:01:40 ***due:*** *Tues, November 17th @ 11:59pm*

Now that you have finish the single-cycle version of our simulator. We are going to improve the performance of our design through two concepts we learned in class: Pipelining and Data Forwarding.

1 Introduction

In this lab, you will implement a timing simulator (written in C) to model instruction/data caches and a branch predictor. While this is not an RTL-level simulation that are synthesizable, it is a higher-level abstracted model designed to allow quick exploration. Operating at a higher level 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 implemented 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 branch predictor as specified

2 The Timing Simulator

Similar to Labs 1 and 2, our only 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 know how well the processor would perform.

3 Microarchitectural Specifications

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 Policy. When a new block is retrieved from main memory, it is inserted into the appropriate set within the instruction cache. If any block within the set are empty, the new block is simply inserted into the empty block. However, if none of the blocks in the set are empty, the new block replaces the least-recently-used block. 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. Finally, note that a redirection that accesses the same block as a pending miss does not cancel the pending miss.

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 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 Instruction & Data Cache

1. Assume that both caches are initially empty.
2. 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.
3. 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.
4. Assume that both caches are initially empty.

3.4 Branch Predictor

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

Gshare. 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 the full 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, the predictor indexes into both the BTB and the PHT. If the predictor misses in the BTB (i.e., address tag != PC 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 branch predictor structures are always updated in the execute stage, where all branches are resolved. The update consists of: (i) updating the PHT, which is indexed using the current value of the GHR (ii) updating the GHR, and (iii) updating the BTB. Unconditional branches do not update

the PHT or the GHR, but only 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:

1. The instruction is a branch, but the predicted direction does not match the actual direction.
2. The instruction is a branch, and it is taken, but the predicted destination (target) does not match the actual destination
3. The instruction is a branch, but it was not recognized as a branch (i.e., BTB miss)

4 Lab Resources

4.1 Source Code

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

1. `Makefile`
2. `refsim`: Reference simulator in machine-executable format
3. `verify`: Script that compares your simulator against the reference simulator
4. `src/`: Source code (Modifiable; feel free to add more files)
5. `src/pipe.c`: Your simulator (Modifiable)
6. `src/pipe.h`: Your simulator (Modifiable)
7. `src/mips.h`: MIPS related pound defines
8. `src/shell.c`: Interactive shell for your simulator (similar to Lab 1)
9. `src/shell.h`: Interactive shell for your simulator (similar to Lab 1)
10. `class-inputs/`: Example test inputs for your simulator
11. `scripts/`: Dependencies for verify
12. `inputs/`: Your custom test inputs (Modifiable; feel free to add more files)

Make sure that your source code is readable and well documented. Please submit the lab on canvas.

4.2 Makefile

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

To compile your simulator:

```
$ make
```

To compile your simulator and verify it against the reference simulator using one or more test inputs:

```
$ make verify INPUT=class-inputs/inst/addiu.x
$ make verify INPUT=class-inputs/inst/*.x
$ make verify
```

5 Getting Started

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.[1] 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 flushing 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 reference simulator (refsim) for all the test inputs that we provide in `class-inputs/`.

```
$ make verify
```

However, your simulator has different microarchitectural outputs (e.g., cycle count) than the reference simulator. Your job is to model the timing effects of the caches, the branch predictor, and main memory so that your simulator becomes microarchitecturally equivalent to the reference 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.

5.3 Tip

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.

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