CS152 Lab 7

Cache and Main Memory

Team Name: I Wish I Was In 150

Aka No Nome

May 10th, 2001

Daniel Patterson (cs152-patter)
William “Billy” Martin (cs152-bem)
Hsinwei “Frank” Chou (cs152-hchou)
Michael Barrientos (cs152-mbarrien)

Abstract:

For the final project we implemented a deep pipeline with 8 stages. We also implemented branch prediction and jump register prediction, and we made a few cache optimizations.

The objective for this lab was to build as fast a processor as possible, that works. Our primary objective for speedup was to reduce the cycle time as much as possible. We also tried to increase CPI by reducing the cost of branches (with prediction) and loads (with cache optimization). The theoretical approach to our datapath yields a setup that should have a very low cycle time, but in practice does not.

Division of Labor:

Each person implemented the component of their choice. Billy did the deep-pipelining, which included designing the ALU and designing the datapath. Frank did branch prediction, which involved keeping track of history, FSMs, and whatever else needed to get a high prediction rate. Dan worked on the cache, optimizing it as much as possible and adding new features. Mike created a JR prediction unit.

Detailed Strategy:

Branch Prediction with history table

There is a branch prediction unit whose job is to solely look up a table of 256 prediction results and output one of them based on the current history of the global history. We decided to use 8 bits index history table (256 slots) rather than the traditional 1024 index slots because we didn't think we would have enough branches to fill up all 1024 slots. The prediction unit takes as input the current global history and outputs a 1 bit predict value based on the finite state machine stored at that history slot in the table. The prediction unit also takes in the actual branch result from a branch and a branch known flag signal that gets asserted when the branch result is known. When branch_known is asserted, the prediction unit takes in the history associated with that prediction and updates the finite state machine of that particular history.

The history unit takes as input the predicted result and updates the current history to reflect the predicted value change. This unit then outputs and passes the history along the cycles so a later instruction can use it. The history has a flag signal called branch_history_shift that shifts a 1 in whenever we predict a branch_taken and 0 otherwise. The history unit needs to be squashed and restarted from time to time because when misprediction occurs, the history will have to be reverted back to the old history, in case branches occurred after branches and the first branch hasn't gotten the result before the second branch requires a prediction. So, when branch miss and renew_branch[7:0 is asserted, we flush the instructions in the pipeline and revert back to the old history.

Optimization of the memory system
The main goal was to lower the cache miss rate and penalty. Several strategies were implemented to allow this. They were:

Fully associative cache with LRU replacement: By using an algorithm to determine the least recently used block in the cache, the most efficient replacement policy was implemented. It was implemented using a matrix of dff's that held the order of used blocks. Associativity, by eliminating conflict misses, the number of cache misses was minimized.

Interleaved memory with burst mode: Burst mode is used for instruction fetch, to get successive instructions as quickly as possible. The latter is so that two word blocks of memory can be fetched simultaneously.

CAM cache: Helps to lower hit time by calculating hit right at cache block.

Victim cache: This enables blocks to be accessed after they have been overwritten in memory without the delay of a memory access. Our implementation operates in one cycle, allowing both a read and write on both the victim cache and the standard cache simultaneously.

8 stage pipeline

We implemented an 8-stage pipeline for our processor. We based it on the suggested 2IF, 1DEC, 2EXE, 2MEM, 1WB model. We tried to design it to allow for the smallest cycle time possible.

Stage IF1

In this stage the instruction is fetched from the cache. Since we are using a CAM cache, this access can be completed in one cycle. The CAM cache is capable of fetching an instruction on a hit (or outputting stall on a miss) in about 16ns. This leaves little room for other manipulations, so no further action is taken in this stage.

Stage IF2

This stage is really more of a decode stage than an IF stage, since the instruction was fetched in the first IF stage. But we called it IF2 anyway, in order to conform with the assignment.

In this stage the instruction is processed to determine if it is a branch, and the branch or jump target address is calculated. Also, some control signals that will be necessary in the future stages are calculated now. Branch prediction and JR prediction is also done in this stage. The critical path for this stage is the time to calculate the branch address (using a 10-bit adder) plus the time to select the NPC (using tri-states).

Stage DEC1

This is the stage when the registers are looked up in the regfile. The regfile is implemented using registers and tri-states, so it should be quite fast. A forwarding mux also exists in this stage which forwards values from the ALU and MEM stages if necessary. Calculating all necessary hazards and stalls also takes place in this stage, and this is most likely where the critical path for this stage lies.

Stage EXE1

In this stage the first half of arithmetic instructions is computed, and logic instructions which have their values ready are also computed. The arithmetic instructions are passed through a carry-save addition/subtraction component and then through a partial carry-lookahead adder/subtractor. This units together take about 11ns, but values that need to be forwarded are ready as soon as they come out of the carry-save component, which means that forwarding can take place in parallel to the carry-lookahead computation and thus does not impact the critical path for this stage. (more on forwarding using a CSA later)

The critical path for this stage most likely involves the logical operations (including shifts), which must be performed as well as selected, and then forwarded.

Stage EXE2

In this stage the second half of arithmetic instructions is computed, and logic instructions that didn't have their values ready by EXE1 are computed here as well. This second-stage logic operation greatly complicates forwarding, since the start and end times for logic operations is variable. But, it eliminates otherwise un-avoidable stalls.

The second half of the arithmetic here essentially just completes the CLA operation and also computes SLT and SLTU values. This information must then be selected by a mux before passed to the next stage, and this is most likely where the critical path for this stage lies.