Design and Verification of Branch Prediction Hardware

Richard Garfinkel

E168b

Spring 2007

Introduction

When designing a pipelined computer processor, one of the most important considerations is the frequency of stalls in the pipeline. An ideal pipeline would result in a processor completing one instruction per cycle. However, a variety of hazards can cause stalls in the pipeline – spaces where no instruction is being executed, thus decreasing the number of instructions per cycle.

One of the most prevalent hazards is the branch. Since branches are used to control the flow of a program and are repeated frequently in loops, branches are a significant component of any piece of software. Unfortunately, the results of a branch are not known until part way through the pipeline, so the next instruction cannot be fetched immediately, reducing the number of instructions completed per cycle. To reduce the number of stalls due to branches, branch prediction methods can be used.

Static Prediction

The simplest method of branch prediction is static prediction. One can predict that branches are always taken or always not taken with minimal hardware. In practice, a static branch predictor should always predict taken, since most branches are taken. The percent of branches not taken in the SPEC benchmark is only 34%.

Dynamic Prediction

The downside to static prediction is that it has a large variance across different programs, and it requires the compiler to optimize for branches being taken. By dynamically adjusting the prediction as the program is running, stalls can be reduced even further.

One-Bit Prediction

A very basic dynamic predictor stores the result of a branch in a table; the next time that branch is called, the result of the previous branch is used as the prediction. For a predictor with 2^n entries, the lowest n bits of the instruction address are used to address the prediction table. In order to update the table after the result of the branch is known, the lowest n bits of the instruction address are also forwarded along the pipeline.

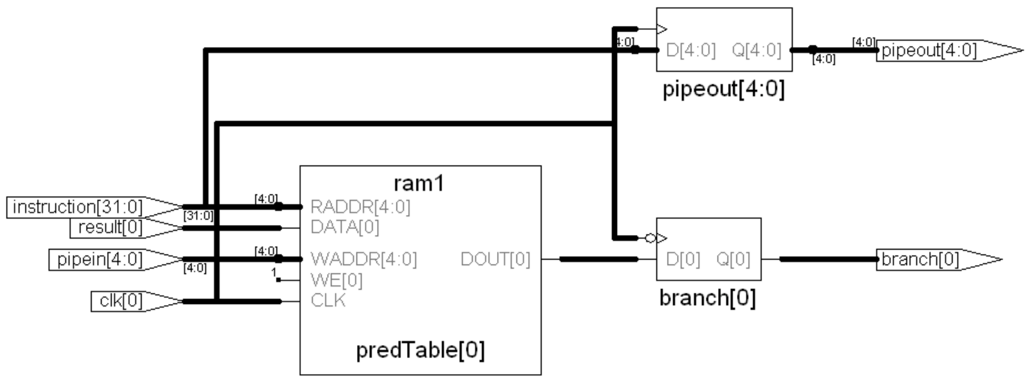

Figure 1: A 32 entry, one-bit, instruction-addressed branch predictor.

Two-Bit Prediction

While a one-bit predictor makes some advances over static prediction, sometimes it changes its prediction too quickly. One instruction might branch three times, then fail to branch once, and then branch three more times. This is especially the case with frequently-called loops. To fix this case, we widen the prediction table to two bits, storing the prediction in the upper bit and the result of the previous branch in the lower bit. The prediction only changes when the past two predictions are incorrect. This also requires two additional bits – the state of the table at prediction time – to be forwarded along the pipeline so that the table can be properly updated.

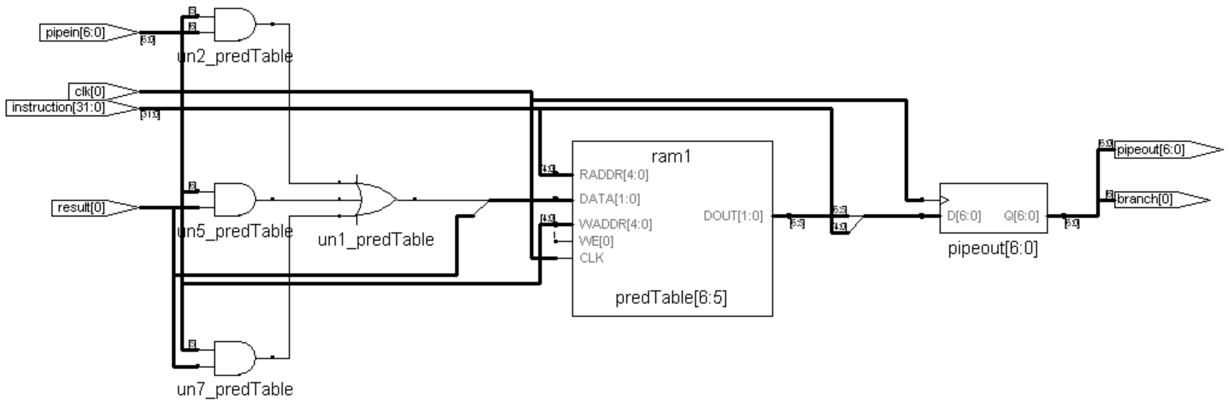

Figure 2: A 32 entry, two-bit, instruction-addressed branch predictor. Note the logical elements used to update the table.

History-Addressed Predictors

The two previously discussed predictors address their tables using the instruction address. As such, their predictions apply locally to the instruction. A different way to address a prediction table is to use a shift register containing the last n branch results to address its table. Since the prediction from such a predictor is not dependent on the instruction address, it is referred to as a global predictor. Global predictors can use both of the table types discussed previously.

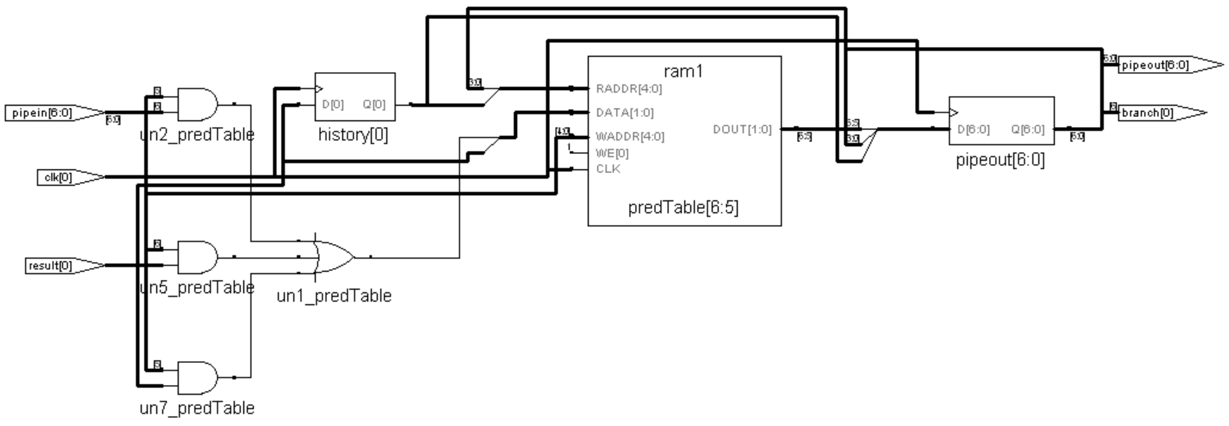

Figure 3: A 32 entry, two-bit, history-addressed branch predictor. Note the shift register holding the branch history.

Tournament Predictors

To make the best use of the merits of different branch predictors, a tournament predictor contains two different predictors as well as a state machine to decide which predictor to use. Typical practice is to use a local predictor and a global predictor, with a two-bit saturating counter to select between the two.A two-bit saturating counter is a state machine with four states: strongly predict with predictor 0 (Strong 0), weakly predict with predictor 0 (Weak 0), weakly predict with predictor 1 (Weak 1), and strongly predict with predictor 1 (Weak 0).

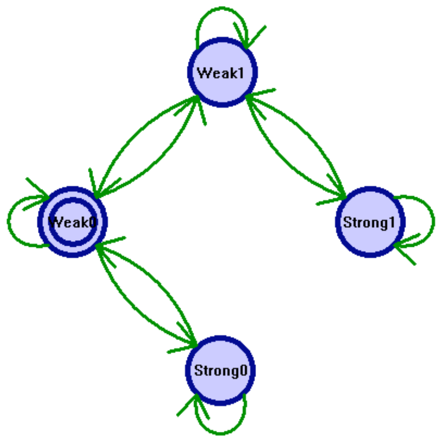

Figure 4: State machine for a two-bit saturating counter. The state moves towards Strong1 when predictor 1 is correct and predictor 0 is incorrect. The state moves towards Strong0 when predictor 0 is correct and predictor 1 is incorrect.

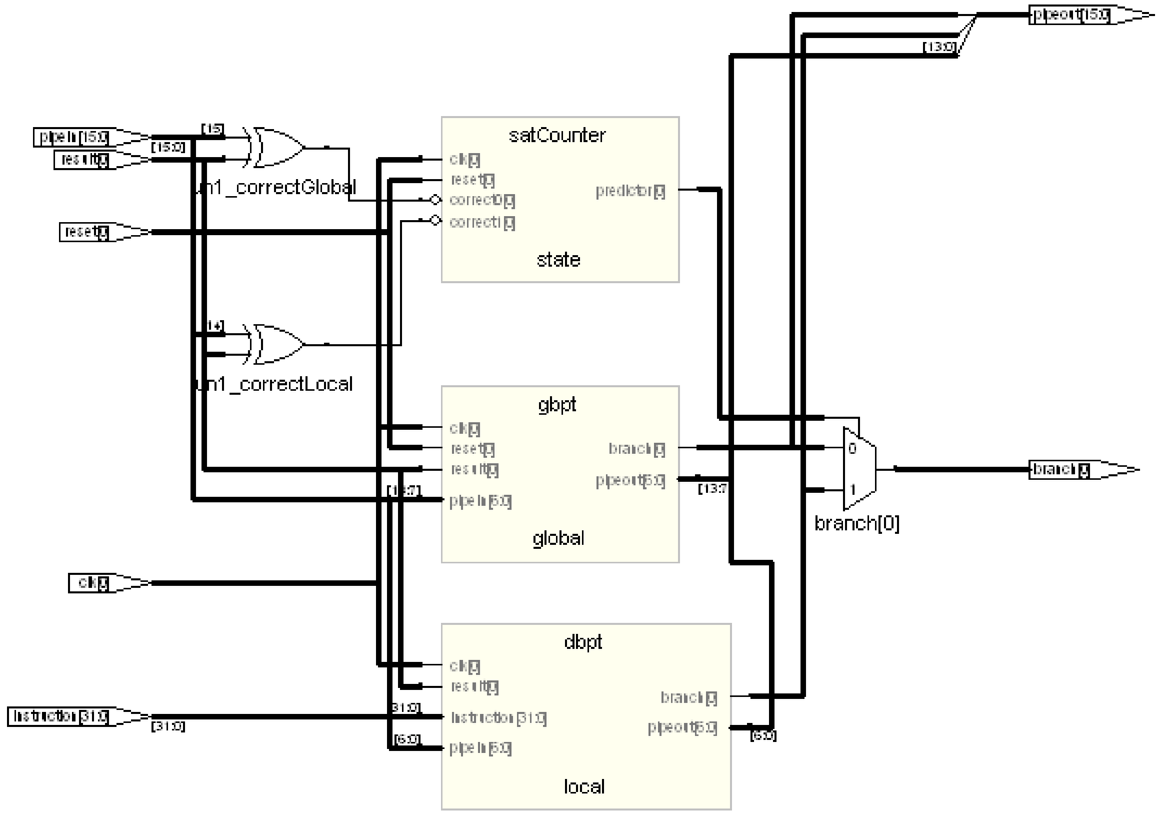

Figure 5: Tournament branch predictor. Note the two independent predictors and the control signal on the mux coming from the state machine. XNOR gates are used to determine if the predictors were correct.

Hardware Requirements

Hardware requirements don't vary greatly between different predictors. The largest component of any predictor is the prediction table itself – it's a bank of RAM. At such a low level, DRAM must be used if the predictors are to be used in high-speed processors. Note that a two-bit predictor doubles the RAM requirement of the same sized one-bit predictor. In the tournament predictor, the independent predictors may use different sized tables, so the tables can be optimized for the individual predictor.

Simulation

After implementing these four branch predictors in Verilog, I tested their accuracy using a list of 1792 branch instructions provided in *Computer Architecture: A Quantitative Approach, 4th Edition* by Hennessy and Patterson. A static predictor running on this code would have been 52 percent accurate. A 32-entry one-bit predictor was 93% accurate. A 32-entry instruction-addressed two-bit predictor was 14% accurate, as was a 32-entry history-addressed two-bit predictor. A tournament predictor using the two two-bit predictors was 51% accurate. Clearly, there is a glaring error here. The tournament predictor performs better than the sum of its component predictors, which is impossible. Also, only the one-bit predictor performs better then static prediction, which independent research suggests should not be the case.

I believe that in simulating the predictors, I improperly simulated the pipeline, and that error caused significant errors in my simulation results.

Conclusion and Suggestions for Further Work

Since the simulation suggests that the implemented branch predictors are not working properly, I cannot confirm the findings of other researchers. Further work on this project would include developing a test bench to debug issues with the simulation and developing new simulation vectors, preferably from a benchmark like SPEC.

References

Hennessy, John L., and Patterson, David A. *Computer Architecture: A Quantitative Approach, 4th Edition.* Morgan Kaufmann Publishers, San Francisco, 2007.

Kuenning, Geoff. Course lecture slides, Harvey Mudd College CS136. Spring 2007.