K7 Branch Prediction

Andreas Kaiser
ak@s.netic.de

Last modified 16. December 1999

1 Preface

This documentis based on publicly available information (or which was publicly available for some time) and personal experience with the AMD CodeAnalyst tool. Correctness cannot be guaranteed. It may turn out to be totally wrong. Corrections are welcome.

Basic knowledge of branch prediction mechanisms is recommended.

2 Branch Detection and Target Address Prediction

Instruction fetches are 16-byte wide, aligned on 16-byte boundaries. The branch prediction provides the address of the next 16-byte fetch block, the position of the first valid byte in the predicted block and the position of the last valid instruction byte in the current 16-byte block. It may predict up to two non-return branches and one return per 16-byte block. Excessive mispredictions may occur if this number is exceeded. So beware of dense branches. e.g. in some implementations of switch statements.

The branch predictor is associated with the instruction cache. In principle, there is one branch selector block associated with each 16-byte cache block. If a cache line (64 bytes) is replaced, all selector blocks for the cache line are invalidated. When the predecoder finds a predictable unconditional branch, the predictor is set accordingly and predecoding continues at the branch target. Conditional branches are not predicted by the predecoder.

For 64KB instruction cache, a complete implementation of this scheme needs in 64K/16 = 4K selector blocks and 8K target addresses, or a total capacity of ˜40KB. But actually there is only enough space for up to 2048 target addresses and the predictor tables are folded twice to achieve this reduction.

The instruction cache is 2-way associative and each way consists of 2048 blocks of 16 bytes (a 64-byte cache line consists of 4 blocks). There are only 1024 selector blocks per way and 1024 target blocks in total, so cache block 1024 shares the predictor blocks with cache block 0, cache block 1025 with cache block 1 and so on. There is a tag bit (A14) stored in a selector block to distinguish both cache blocks which map to the same selector block. If the tag bit does not match, there is no prediction and fetching continues sequentially.

Both selector tables share a single target address table. So while there may be up to 4 non-return branch selectors in a selector block set (the two selector blocks in both ways having the same index), the prediction is limited to 2 branches per set and target address mispredictions occur if there are more. Since return addresses are predicted using a 12-entry return address stack, return instructions do not interfere.

For simplification, figure 1 shows 3 distinct branch selector fields in each selector block. Actually they are encoded as shown in figure 2 to reduce the time required for evaluation. There are 9 selectors per selector block, 2 bits each, which indicate the prediction for an instruction fetch which starts at the respective offset. All predicted branches except for the single-byte return (opcode C3) are at least 2 bytes long, so there may be at most one branch ending in any pair of bytes. The selectors for offsets 0 and 1 have to be distinct because while a branch ending at offset 8 is encoded in all selectors up to offset 7 but not in the selector for offsets 8 and 9¸ a branch ending at offset 0 must be present in the fetch block where branch ends, not just in the preceding fetch block. Single-byte return instructions impose a problem as their location is ambiguous. If a nonsequential fetch (a branch target fetch or a misprediction recovery) starts at the location of a return instruction and the offset is even and not 0, the return instruction is mispredicted.

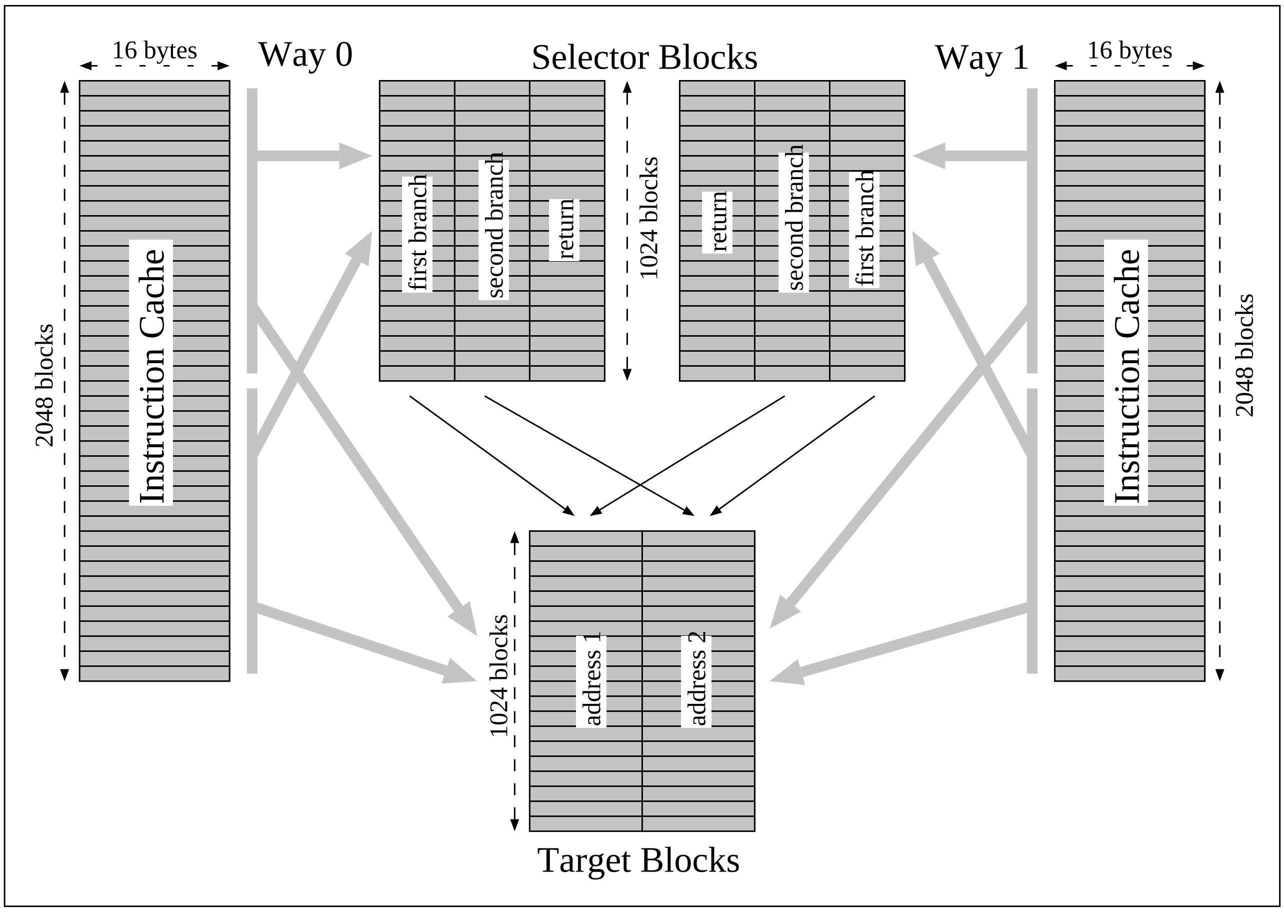

Figure 1: Branch Target Buffer

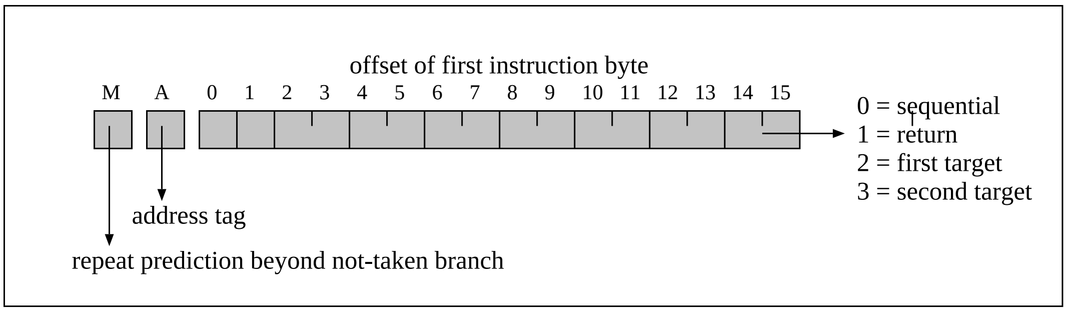

Figure 2: Branch Selector Block

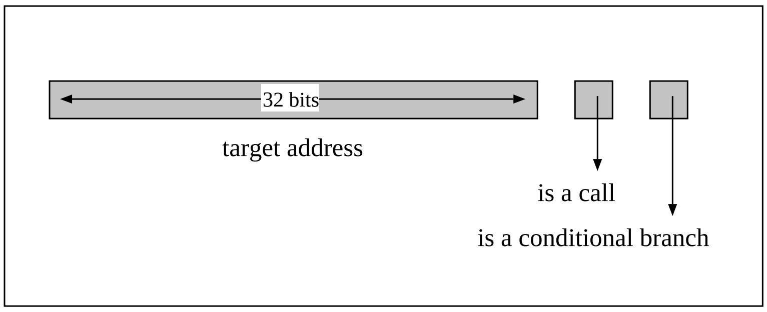

Figure 3: Branch Target Block Entry

This prediction mechanism is very fast because no correlations between branch positions have to be considered at prediction time, and if selector presence implies *taken* (see below), even the outcome prediction does not cause further delays. In some preceding designs, AMD ran the prediction to the instruction fetch stage, to avoid any delay for taken branches. However since one of the key design goals of the K7 was clock speed, the predictor actually runs in the second pipeline stage. So there is a single cycle fetch bubble for correctly predicted taken branches. But there is a small queue between fetch and decode and, on average, the fetch bandwidth exceeds the decode bandwidth. So if there is no more than one taken branch per 2..3 fetch blocks, this bubble is likely to be hidden.

3 Conditional Branches

The initial prototype used a fairly simple mechanism for *taken / not taken* prediction of conditional branches. In the target block entry (figure 3), there was an additional bit for conditional branches, which can be thought of as the LSB of a crippled saturating 2-bit counter predictor having 3 states instead of 4. A previously undetected taken branch entered the predictor in state 2 (*weakly taken*) and if it was still taken on the next occasion the state was incremented to 3 (*strongly taken*). If a branch in state 2 was mispredicted, the corresponding selectors were removed from the selector block, to predict *not taken* next time. However this situation was identical to a previously unseen branch, so there was no *strongly not taken* state as in a regular 2-bit counter predictor. No matter how often a branch was not taken, on the next time it was taken, it entered the predictor as *weakly taken,* to be predicted *taken* next time.

Somehow AMD must have found out that this predictor was far from being state of the art, especially for a machine having such a long pipeline. And the K6 has an excellent history predictor, so why not use it? Well, the scheme described in the previous section works best if the predictor can tell whether a branch is predicted taken **next time**, to be able to evaluate the prediction at the time the selector block is updated in case of misprediction: if the selector is present, it is predicted *taken*, otherwise it is predicted *not taken.* However the K6 predictor does not have this property, the next outcome of a conditional branch cannot be predicted at time of misprediction because it depends on the outcome of the 9 branches executed before.

Nevertheless AMD decided to use the history predictor of the K6 in the production version, but reduced the global history from 9 to 8 bits to conserve chip space. The prediction history bit in the target block entry changed its meaning into *is a conditional branch* and a selector hit does no longer imply a taken branch. Instead the outcome predictor has to be consulted for a conditional branch and if there is some other branch in the fetch line beyond a not taken branch, the predictor repeats the prediction process in the next clock cycle, starting beyond the first branch.

To avoid a fetch bubble for every not-taken conditional branch, the branch selector block contains additional information on whether there may be some other branch beyond a conditional branch: if the M bit shown in figure 2 is reset, a not-taken conditional branch predicts *sequential.* However I am uncertain about the exact conditions under which the M bit is set, maybe it is just an indication for the number of branches encoded in the selector block (1, >1).

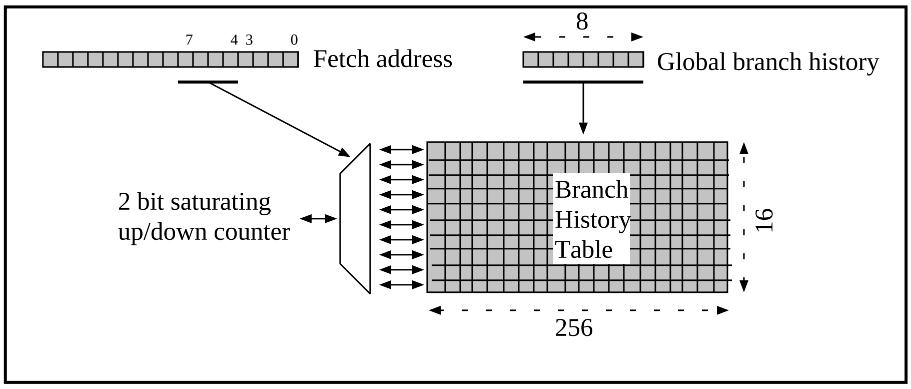

Figure 4: Branch History Table

References

[1] "Branch Selector Prediction", AMD, US patent 5,954,816. Describes the branch selector encoding.

[2] "K7 Challenges Intel", Microprocessor Report, Volume 12 number 14. Has a few words on the branch predictor of the initial prototype, as shown in the Micrprocrssor Forum 1998.

[3] “AMD-K7 Processor Optimization Guide”, revision C-4, part of the initial public release of the AMD 3DNow! SDK 3.0. Later revisions do not contain any information on the branch prediction.

[4] Voice recording of the dinner presentation by Dirk Meyer (AMD), June 1999.

[5] Microprocessor Report, 23. August 1999. Apparently based on the dinner presentation.

[6] “How to optimize for the Pentium family of microprocessors”, Agner Fog, 1999-01-18, http://www.agner.com/assem. Describes the branch prediction used by Intel processors.

[7] “Method and apparatus for implementing a branch target buffer in CISC processor”, Intel, US patent 5,903,751. Describes a BTB which looks somewhat similar to the one used by the Intel P6 family processors.