HW2 Part 2 Report
Analysis Report of Aquila's Branch Predictor

Tzu-Han Hsu
dept. Computer Science
National Yang Ming Chiao Tung University
Student ID: 081358

***Abstract*—We verified improvements in IPC by adding branch prediction unit to Aquila SoC. We found in Aquila's default BPU, about 90% of all branches are not recognized and the default not taken decision is taken. Among those recognized branches, BPU only guess 55% correct when it comes to forward branches. Our implementation of two-level branch predictor performing worse than one-level BPU, only score77.297 in Coremark. Where one-level BPU scores 84.837. 3-bit saturated counter fits the two-level branch predictor better than 2-bit counters, improving Coremark score about 1.5.**

***Index Terms*—Branch prediction, RISC-V, Aquila SoC**

I. How branching statistics were collected?

The branch predictor mechanism works as follow: The Program counter asks BPU whether there's an existing entry. If an entry is found in the table, program counter hands on the decision of whether to branch to the branch predictor. BPU looks up the table to decide whether to take the branch according to its previous history. If no entry is found, the program counter assumes the branch will not be taken and moves the program counter to the next instruction. During the execute state, whether the branching shall take place is answered by the calculation of specific register values. The EXE stage sends back signals to the branch prediction unit to update the BPU table entry.

To analyze how the branching mechanism works, counters must be set. The counters for analysis in this work are placed at the execute stage, for the following reasons:

1) Execute stage is the first stage to certify the assumption of branching or not. There are signals telling whether the branch is miss predicted, the original decision could also be obtained from the pipeline registers.
2) Decode stage has decoding signals telling which kind of branch it is, a forward or backward jumping branch, or an unconditional jump.

By referencing the branching address extracted from the instruction and the instruction type, we have a clear view which kind of branching instruction it is. These branching type signals are best collected in the decode stage, then send to the execute stage through pipeline registers.

II. Analysis of branching statistics

From Table I, turning on the branch prediction unit improves the Coremark score, from 74.965 to 84.837. The improvement shows on the total cycle counts on executing the program. Without the branch prediction unit, Aquila has to spend extra 78,263,922 cycles to complete the program. From Table I, we may conclude that branch prediction unit in Aquila do improve the IPC of Aquila.

TABLE I
Clock cycles spent on each Function

	BPU on	**BPU off**
Coremark	84.837	74.965
total cycles	596,898,253	675,162,175
branch / jump cycles	138,414,146	151,411,451
branch /jump %	23.198%	22.425%

TABLE II
Clock cycles executed by each Instruction type

	F.W branch	**B.W branch**	**uncond. jump**
total cycles	70,613,032	54,811,933	12,989,181
stalled cycles	18,284,651	10,473,042	2,672,428
stalled %	25.894%	19.107%	20.574%
BPU Miss	67,333,144	47,873,053	12,983,850
BPU Miss %	95.355%	87.341%	99.959%
Branch	130,703	413,979	146,012
Not Branch	67,202,441	47,459,074	12,837,838
BPU Hit	3,279,888	6,938,880	5,331
True Positive	1,553,786	1,205,488	-
True Negative	276,925	5,047,570	-
Flase Positive	3,661	609,476	-
False Negative	1,445,516	76,346	-
BPU Accuracy	55.816%	90.116%	100%
Overall Accuracy	97.376%	97.994%	99.876%

Table II shows the statistics collected categorized by different branching types. Most of the branch jumps to a larger PC address (forward jump), unconditional jumps like JAL and JALR are the least type of branching instructions, sharing only 9.3% of the overall branch/jump instructions. From HW1's analysis, we know that branching and jump instructions account for 23% of all instructions.

Another interesting discovery is most of the branch address are actually not indexed in the Branch Prediction Unit. From Table II, we may observe that over 90% of the branching instruction do not appear in the BPU table. The branching decision at most of the time is made up by the program counter, who violently assume branch not taken. Table II shows how high percentage of these unregistered branch are not taken. A reasonable explanation is that these unheard-of branches indicates control sequence that are seldom used, and do not work most of the time.

Do those indexed branches, those branches whose address appear in the BPU table, do a better job? Table II shows how the default one-level bimodal branch predictor do:

A. *Conditional forward jumps*

Bimodal branch predictor handles Forward branching very poorly, 55% of accuracy is collected running Coremark program. True Negative and False Positive rate are very low, indicating that most forward branches recorded in BPU would take the branch. This explains why false negative rate in forward branching is also very high, guessing branch not taken in forward jumping is apparently not a good idea. Forward branches are the most common branching type, and the default BPU in Aquila has difficulty predicting it. The overall branching accuracy (include those not in BPU) is 97.376% in this category.

B. *Conditional backward jumps*

About 90% of the backward jumping branch is predicted correctly. According to the statistics, most backward branches are not taken, the ratio is about 4 : 1. The statistics seems to indicate that conditional backward branches behave more predictably when compared to forward jumps .If those unrecorded backward jumps are included in our calculation, about 98% of the backward branch are correctly predicted.

C. *Unconditional jumps*

Unconditional jumps are different from conditional jumping, branching recorded under the BPU memory would reach 100% accuracy by default. However, unconditional jumps are relatively uncommon in branching/jump instructions, only about 10%. Unconditional jumps are those instruction you would hope to keep in BPU, because they always points to the correct direction.

The statistics indicates that the native Branch Prediction Unit in Aquila is performing quite poorly. Most branching instructions are not recorded in the BPU and are guessed not taken. Surprisingly, the guess is correct most of the time. The records in the BPU are those branches that do not perform regularly, especially conditional forward branches.

III. Implementation of a Two-level branch predictor

In order to gather global information, a global history table collects the branching decision of all recent branches regardless of their respected program counter. The table is essentially a left shift register tuple, shifting the latest branching decision from LSB. In our design, the global history table is a 5-bit left shift register.

Local information is collected by the local history table. Branching decisions recorded in the local history table are separated by program counter, so the size of the table would affect the granularity of the branching record sampled. In order to reduce the table size, our implementation limits the local history table to size 32×5. The significant 5 bits of the PC, bit 14 $\sim$ 10, are used as index to the table. The default BPU uses the full PC to record the branch address, which is a great waste of memory. Since bit 15 $\sim$ 31 are actually all zero when Coremark runs, we shall omit storing them. Note that gshare engages both global and local information by exclusive or the two value, so their length must match.

A two-level branch predictor incorporates a branch likelihood table to make the decision of branching or not at the fetch stage. The size of the branch likelihood table is once again a trade-off between size and granularity. In our experiment, we set the size to 16 to balance between table size and effectiveness. Let G be global history table size and S be saturating counter size. The entry size of the branch likelihood table (BLT) is:

$$size(BLT) = 2^G \times S$$

the formula shows the global history table size, which is also the size of each entry of the local history table, is exponentially related to the BLT size. It infers that by enlarging the local history table to store more branching history is theoretically spatially inefficient, because an even larger branch likelihood table must be stored. The sweet spot of the size of record to hold is a nice topic to further investigate.

TABLE III
Performance of all BPU models

type / size / counter	Coremark	total cycles	stalled cycles
off	74.965	675,162,175	88,262,359
1-level / 32 / 2	84.608	598,519,532	88,411,359
1-level / 64 / 2	84.837	596,898,253	88,408,431
1-level / 64 / 3	84.78	597,298,915	88,408,441
2-level / 32 / 2	75.866	667,174,459	88,323,175
2-level / 64 / 2	75.874	667,343,189	88,323,193
2-level / 64 / 3	77.296	654,880,208	88,408,015

IV. Analysis of BPU performances

We are interested in testing how three different attributes of a BPU design would affect the Coremark performance of Aquila. One-level predictor against two-level predictor, different BPU record size, and the size of the saturated counter. The accuracy of each BPU unit should incorporate in the comparison to provide a better insight of how distinct models work. Sadly, there is some technical issues so only Coremark score and total cycles ran are shown here. Despite the fact that the overall accuracy of our prediction unit is not shown, we may say that the BPU design is better if and only if it has a better Coremark score in our experiment since all other factors are controlled and stayed constant.

Table III lists all experimented results. An important and instinctive conclusion is that all BPU models works better than without one. Even the worst performing model with 2-level BPU, 32 entries of branches and 2 bit saturated counter is 1 point higher in Coremark score comparing to model without a BPU.

A. One-level BPU v.s. Two-level BPU

One-level branch prediction collects only recent history from the execution of a local area, whereas two-level branch prediction has a larger vision. Two level prediction requires more resources to implement, structures like local history table and pattern history tables are large 2D register arrays. So theoretically, a two-level branch predictor should perform better than a one-level branch predictor.

From Table III, our experiment seems to show a different result. Two-level branch predictors seems to perform worse than one-level branch predictors. The best performing one-level BPU is with 64 entries of branch table and 2-bit saturated counter, scoring 84.837 in Coremark. Two-level BPU with 64 entries of branch table and with 3-bit saturated counter scored only 77.296. The data collected from our experiment contradicts with the theory.

Two-level branch predictor is much more complicated than one-level branch predictor, so fine tuning becomes more important. Grid search shall be taken to find the best configuration like global branch history size, local history table and pattern history table entries. I believe after some fine-tuning, two-level branch predictor models would eventually outperform one-level branch predictor.

B. BPU branch table size

Table size of the BPU is related to the number of PC address that was confirmed as a branch instruction. The larger the size of the BPU, the more address could be stored thus larger portion of the branch executed could take advantage of the branch predicting mechanism. The entry size of the BPU table is linearly related to the overall memory usage in the branch prediction unit module. In our implementation, a 2D register array is used to record the branching address and the block ram is used to store the branching address.

We are interested with the relationship between performance and the BPU branch table size. Experiments with configurations of 32 entries and 64 entries are conducted. From Table III, one-level predictor with 2-bit saturated counter and two-level predictor with 2-bit saturated counter are tested. We may observe that by enlarging the table size, better Coremark score could be found. If 1-level branch predictor is used, enlarging the BPU table size from 32 to 64 would enhance the Coremark score by about 0.2, from 84.608 to 84.837. If 2-level branch predictor is set up, 32 more entries are not very beneficial, only from 74.866 to 74.874, up by 0.01.

If we have abundant space to build the BPU unit, enlarging the BPU branch table is not a bad idea. Since a bigger table is always more powerful than a smaller one. We have to bear in mind though, the law of diminishing marginal utility would soon catch up when the table size is close to the amount needed by the executing program.

C. saturated counter size

The saturated counter records how likely the branch will be taken and the default size of the saturated counter is 2-bits. The original implementation of the counter in Aquila is by finite state machine. However, by implementing a 3-bit saturated counter, FSM may not be a good idea. A behavioral counter is written in the RTL level since it's more convenient for the designer.

The initialization state of a 3-bit saturated counter is a little different from a 2-bit saturated counter. Under a 2-bit saturated counter BPU, if the branch is first observed taken, state 2'b11 is initialized. In the 3-bit saturated counter, state 3'b110 is initialized. A 3-bit counter BPU needs more incorrect predictions to change the branch decision, so if the initialization value is 3'b111, we may suffer from more incorrect branches. As if the prediction is correct the next time, we could reach 3'b111, indicating the branch is almost always taken. How the initialization could affect the performance of the branch predictor is another topic to further dive in.

Would switching to 3-bit saturated counter lead to better performance? Table III shows the statistics collected from the experiment. One-level branch predictors and two-level branch predictors with 64 BPU table entries are tested. We may observe that adopting a 3-bit saturated counter compromises the performance of the one-level branch predictor, dropping the Coremark score from 84.837 to 84.78. As for two-level branch predictor, adopting 3-bit saturated counter boosted the Coremark score from 74.874 to 77.296. The experiment tells us that altering the saturated counter size could improve or lower performance quite a bit. Our experiment shows no conclusion of 3-bit saturated counter is better than 2-bit. Seems like different program running on different BPU has its unique optimum saturated counter size.

V. CONCLUSION

Despite the existence of a branch prediction unit. Most branch instructions at the fetch state are still not recognized by the BPU. These branches are automatically assuming branch not taken, and the mechanism works quite well. The default one-level 2-bit saturated counter BPU is not doing a good job when it comes to forward branching, guessing only 55% correct. By switching to two-level branch predictor, the performance in our experiment actually dropped, different from what we expected. 3-bit saturated counter fits the two-level branch predictor model, raises the Coremark score by about 1.5. Maybe further fine tuning shall be done to reveal the true capability of a two-level branch predictor.