ARCHITECTURE-LEVEL POWER OPTIMIZATIONS FOR THE ADPCM BENCHMARK

Author info

Abstract: *The goal of this project is to … The chosen metric is …*

1. Introduction

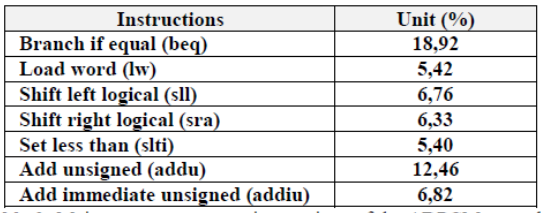
…

2. Benchmark

The benchmark chosen for this project is the ADPCM decoder, the following is a briefly explain of how it works. Adaptive Differential Pulse Code Modulation (ADPCM) is an audio coding technique that is widely used throughout the telecommunications industry. It works by calculating the difference between two consecutive samples in standard pulse code modulation (PCM) and codes the error of the ‘predicted’ next sample increment (from the previous sample increment) to the true sample increment [4].

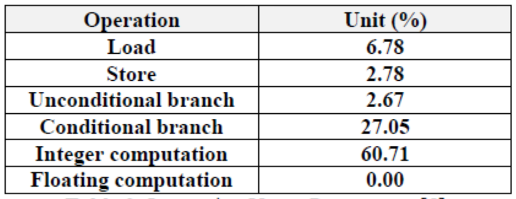
Only the decoding phase has been part of the analysis, the source code is small allowing a deep exploration in order to identify the critical operation and resources consumption. The benchmark takes a file as an argument in ADPCM (clinton.adpcm) and gives it as an output in a PCM file (out.pcm). After analyzing the source code, we found that the main resource consumptions are integer operations and conditional branches (See table 2).

After that analysis, we ran the sim-profile simulator using the parameter (–*iclass*) in order to verify our previous conclusion and to obtain a general summary of all the instructions, as shown in table 3.



Instructions	Unit (%)
Branch if equal (beq)	18,92
Load word (lw)	5,42
Shift left logical (sll)	6,76
Shift right logical (sra)	6,33
Set less than (slti)	5,40
Add unsigned (addu)	12,46
Add immediate unsigned (addiu)	6,82

Table 2: Main usage percentage instructions of the ADPCM assembly code [5].



Operation	Unit (%)
Load	6.78
Store	2.78
Unconditional branch	2.67
Conditional branch	27.05
Integer computation	60.71
Floating computation	0.00

Table 3: Instruction Usage Percentages [5].

As shown in table 3, integer computations (60,71%) and conditional branches (27,05%) are our main critical resources, confirming our expectations. The load operations (6,78%) are also considered, but with less efforts. Although the floating computation (0%) has no effect usage percentage in our code, such data was very important to be known since the beginning of our simulations, because we could also minimize the effect of the floating ALU on our hardware architecture.

3. Cost Function

…

4. Design Space Exploration

…

4.1. Specifying the Branch Predictors

Considering that almost 30 % of our benchmark is branch resources, we have spent a special attention on them, especially on all branch prediction techniques the WATTCH simulator provided us. The branch predictor acts at the ID stage. The performance of a branch prediction depends on its accuracy (misprediction rate) and the time wasted executing non-useful instructions.

From our first investigation on branch predictor (See Figure 3 and 4), we could see that the Perfect branch predictor minimized our cost function considerably by achieving the lowest CPI. The problem is that this technique is applied for ideal application, so we did not considered the Perfect branch predictor.

Moreover, we also eliminated the *taken* and *not taken* branch predictors, since their results were unsatisfactory (very high CPI). It occurs simply because those techniques are trivial and inefficient in our benchmark. Secondly, considering the fact those techniques do not use a good prediction in managing our 27% of conditional branch instructions, the results are time wasted executing non-useful instructions, and higher CPIs. In taken prediction, for example, each conditional branch is executed. Therefore, at the ID stage, we know that a branch instruction is already available, so we can start to address it and to store it in pipeline sequence. However, all of this work is in vain, when at the EX stage, this instruction has not taken place yet, meaning time wasted.

On the other hand, the other branch prediction techniques demonstrated almost the same cost function and CPI average. For this reason, we have chosen 2 predictors with the best cost function to our design space exploration, as presented in Table 5. Those are the combined and the 2-level prediction, which allow dynamic branch prediction. Both techniques use branch prediction buffer to determine whether the branch was taken or not. In particular, we noted that the combined prediction with 256 entries in the buffer had exactly the same CPI, but lower power consumption, than the other combined predictors with larger entries. Regarding the 2-level prediction, we could also observe that the better configuration was the type with the adaptive structure (2lev 8 65536 8 0). This is the PAp predictor, which defines a table with adaptive branch prediction parameters [3]. The PAp was selected due to the very low CPI it provided, even though its power was as high as the perfect predictor. We had also in mind to investigate at least two different and previously set branch predictors since our number of simulations is limited.

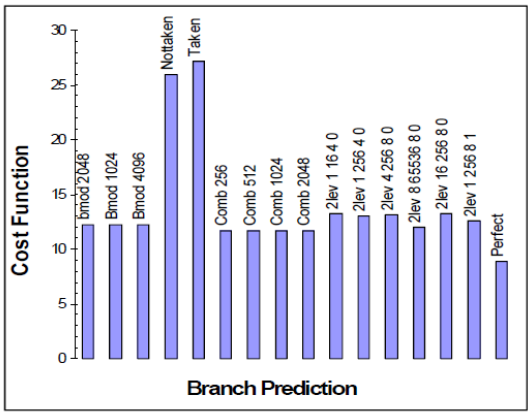


Figure 3: Cost Function vs Branch Prediction.

Predictor	CPI	Power	Cost Function	Cost Function Reduction (%)
Bimod 2048 (Default Conf.)	0,6920	17,7367	12,27	------
Comb 256	**0,6317**	**18,4905**	**11,68**	**4,83**
2lev 8 65536 8 0	**0,6006**	**19,9535**	**11,98**	**2,36**

Table 5: Cost Reduction (%) of the chosen branch predictors.

Finally, from this preliminary branch prediction analysis, we could eliminate 3 branch predictors (taken, not taken and perfect) and notify that the dynamic branch predictors have almost the same cost function level. Since our goal is to minimize our cost function, our priority was the branch predictors with the highest cost reduction. …

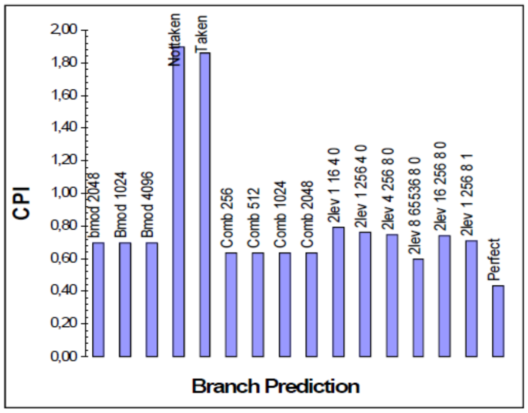


Figure 4: CPI vs Branch Prediction.

4.2. …

…

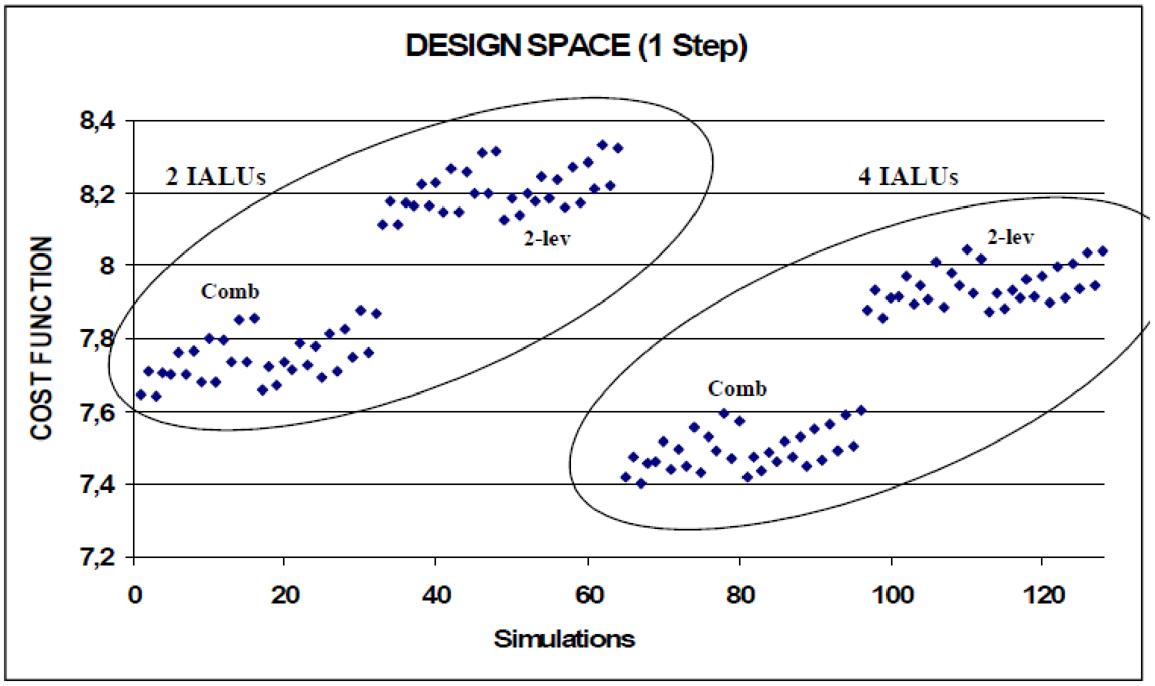


Figure 8: Design Space (Focus on IALUs, Branch Predictors and Caches).
OPTIMAL CONFIGURATION: BPRED:{Comb 256}; IALU:{4}; IL1: {32}:{16}:{1}:{l}; DL1: {64}:{16,32}:{1}:{l}; UL2: {1024}:{32:{1}:{l}