CVA6 Student Contest - Télécom PaRISC

In this contest, we were tasked with improving the CVA6 architecture in two main aspects: increasing the maximum clock frequency and the coremark score.

We decided to mainly focus on adding features rather than optimizing bits of the current design, since we deemed the former to be easier than scanning the entire codebase and looking for potential optimizations.

Results

First and foremost, here are our final results:

1. Reference frequence, final frequency

Reference	Final
42.9 MHz	43.9 MHz

(The details of how we obtained these values are given below).

2. Reference CoreMark/MHz, final CoreMark/MHz

Reference	Final
112.2155	111.8707

3. Reference FPGA resources, final FPGA resources (LUT, FF, DSP)

	LUT	FF	DSP
Before	14807	9286	4
After	15059	9291	4

4. Number of lines added/deleted/modified.

233 added lines and 2 deleted.

Approaches

Improving the coremark score means improving the instruction throughput of the whole pipeline. We took three different approaches (we mainly borrowed our ideas from modern processor architectures):

- prefetching data
- reordering decoded issues to avoid stalls between ID and IS stages
- out-of-order issuing in IS stage

We only obtained positive result with the first approach unfortunately. Nonetheless, we still think the other ones are worthwhile.

Successful approach: prefetching data

Introduction

Prefetch is a mechanism by which data is cached before it is requested by the program. This requires a good prediction in order to do so effectively and not waste ressources. Luckily, memory intensive workloads often involve accessing/scanning arrays which creates patterns easy to predict.

Design

Prefetch module

In order to integrate a prefetch easily into the already existing architecture, we decided to put a module between the core and the cache. This has the benefit of being transparent on both the core and the cache subsystem, and therefore not involve any kind of modification on both parts, except routing the request bus to the prefetch module instead of a direct connection. However, this approach is less efficient than a tight integration into the cache as there are wasted ressources used to transmit the content of the prefetched addresses to the prefetch module, data which is thrown away as it is not needed at the time by the processor.

Module logic

Prediction logic

Whenever the processor makes a load request to the data cache, the prefetch module (thereafter shortened by PF) logs the index addressed and the tag. The difference between the last two indexes accessed is repeated - beginning from the last access - to generate 8 predictions. The difference used is adjusted to account for the cache line size and ensure all predictions have chance to generate a cache miss (except the first for a reason explained further bellow).

It's the generation of a cache miss that makes the prefetch do the job of pulling predicted data from the memory to the cache.

Preemption logic

When the core makes a load request, the PF increments a counter, and each cycle, if the core doesn't make a request, the counter is decremented. This prevents the prefetch from making an attempt while the processor is likely to make a request during the attempt, therefore losing cycles in the execution, as it has to wait for the completion of the cache miss.

There is also a counter counting the numbers of predictions confirmed by the next access. This is to prevent the PF from making attempts while we are still unsure that the pattern holds.

The PF takes control of the bus only when it isn't in use, when the core isn't likely to make a request following the preemption and when it is confident enough.

This creates 4 parameters controlling the likelyhood of attempts. The increment and decrement constants for the memory use accumulators, and the 2 thresholds for memory use and confidence.

These constants have to be tuned in order to result in an optimal triggering of the prefetch.

When all predictions have been prefetched or as soon as possible when the core makes a request, the PF gives control of the bus back to the core. A confidence limiter has been implented in order to not prefetch all predictions if the confidence is too low.

Results

The coremark score with the constants tuned as best as they could be by humans and our limited means gives poor result with loss of score from 112.2155 to 111.8707. Some attempts going as low as 110.7143. However we can see that coremark isn't very memory intensive, at least not in a way usable for such a simple prefetch. (numbers obtained by simulation)

However, we made a surprising discovery by mistake in synthesizing the prefetch branch instead of the master branch to establish baseline numbers.

If our understanding of frequency computations by Vivado is correct, the prefetch module improves the frequency : from WNS = 1.705 ns resulting in minimal period = 25-WNS = 23.295 ns which implies a frequency of 42,927 MHz, we go to WNS = 2.223 ns, minimal period = 25-WNS = 22.777 ns, so a frequency of 43.9039 MHz.

This result is suprising and may be caused by optimisations made by the synthesis suite.

This results in a net performance gain from 4817149504.02 coremarks/time to 4911564276.82 coremarks/time, so a gain of about 2%.

Unsuccessful approach: Out-of-order issue / superscalar architecture

The CVA6 features a scoreboard system and independent FUs. This means that this architecture is able to execute multiple instructions independently when possible. However, instructions are issued in-order. This design results in stalls in the scoreboard whenever one instruction can't be issued immediately because of busy FUs.

This modification aims to fix this isuse by allowing the scoreboard to issue instructions out-of-order (if possible). It could even be improved further and make it possible for multiple independent instructions to be issued at once during the same cycle. We obtain in the last case a superscalar architecture.

Steps

We intended to implement this design with these steps:

- decoupling inserting decoded instructions into the scoreboard and issuing instructions
- separately compute the `rd_clobber` signals
- separately compute the forwarding signals
- separately read operands and push instructions to the right FU

If done correctly, the preexisting code would handle the instructions commit part.

Although untested, the last step was the only truly implemented part.

Implementation difficulties

Despite our many attempts, the first step turned out to be too difficult for us. Decoupling the inserting and the inserting is easy, but making sure that the pipeline doesn't somehow get stuck is not. Flush signals turned out to be problematic as they did not always leave the scoreboard (or the surrounding modules) in clean states.

Another unsucessful approach, part 1: Reorder Buffer

Introduction

The reorder buffer is a module located between the instruction decode stage and the issue stage. It can swap two consecutive instructions which is interesting in two situations:

- an instruction requiring multiple cycles in the EX stage comes while its FU is busy (e.g, load/store, floating point operations)
- chaining MULT instructions using the mult unit (divisions and multiplications block all FLUs except the MULT unit itself)

Design

`instr_reorder` module

The reorder buffer has two states:

- empty: the module only transmits signals between the decode and the issue stage and nothing else
- full: whenever an the scoreboard stalls (scoreboard full or unavailable required FU), the reorder buffer stores the decoded instruction in a buffer. Then, during the following cycle, if the buffered instruction still can't be issued and the newly decoded instruciton can, the latter is issued if possible.

Deciding when to swap

Before swapping two instructions, the module first makes sure that swapping will not create any hazard (by swapping a branch instruction or breaking dependencies between instructions). Then it checks if it is profitable (e.g. the swapped instruction is a stalled load/store).

Returning to empty mode

When empty, the buffer is transparent and waits for the scoreboard to stall. If a branch instruction is decoded, then the module forcibly empties its buffer and returns to empty mode. This feature avoids unnecessary fetches in the IF stage in case the branch predict turns out to be wrong.

Efficiency

The reordering buffer efficiency is a bit unpredictable as it depends on the optimizations performed by the compiler. We were told for instance that compilers tend to place load instructions as far as possible, which results in blocks of load instruction. On different coremark binaries, we witnessed both slight improvements or slight regressions (the latter might be due to changes in the rhythm at which the instructions are fetched, resulting in cache misses).

The main problem of this module is the faster instruction fetch rate: fetching more results in more useless memory transactions in case of a wrong branch predict. However, we found out that IF wasn't always requesting data from the I$ (null `req` signal). We decided to take advantage of this and prefetch instructions from both possibilities in case of a branch.

Part 2: Additional Branch Prefetch

Description

The idea of this modification of the frontend is to take advantage of unsused cache time to fetch into the cache instructions that are predicted as unused by the branch prediction. Therefore, if the prediction is correct, no time is lost (as the time used is originally wasted time due to the instruction queue being full), but if the prediction is wrong, the next instruction will already be cached.

Design

This feature appears as an additional register "unpredict_addr" inside the frontend module that reminds the last address left by the branch prediction. This address is given to the cache only when the "instr_queue_available" bit is false, in order not to slow down the normal functionning of the frontend. When a rising edge is detected on the "instr_queue_available", the fake request is killed.

Efficiency

This feature has lowered the coremark score.

Moreover when combined with the reordering buffer, the simulation never finishes, therefore one of the two module must be faulty. The branch prefetch is more likely as we have a better understanding of the reordering buffer.

We hypothesized that the I$ removed relevant cachelines when prefetching since it uses a random replacement policy. This led us to replace it with a LRU policy.

Part 3: LRU Policy for ICache

Introduction

The initial policy for entry replacement in the instruction cache was a random policy. We have replaced it by a Least Recently Used policy. So each entry in the cache has an associated age (time since last usage) in order to know which entry to discard when a new entry is entering the cache.

Design

Ages

The ages are stored in registers. An implementation using sram would probably have been better, however we needed to both read and write an age during the same cycle therefore the provided sram wasn't suitable, and we hadn't enough time to create a custom one.

Age modification

The feature is designed so that all values possible for the parameter age are given to an entry once and only once. So when an age is set to 0 only the ages that were strictly inferior to this one will be increased. Therefore, there will always be one and only one entry with the maximum age, and this entry will be the one discarded in case of entry replacement.

Age reset

The age of an entry is set to 0 whenever the entry is fetched from the memory, or when a cache request from the frontend hits on the entry.

Efficiency

The implementation of this policy made no difference for any of the benchmarks.

However, since coremark is rather short (we estimated that it could almost fit inside the 8kB instruction cache), it may be possible that the LRU policy could not prevent cacheline replacement because no cacheline ever get replaced to begin with.

Conclusion

Unfortunately our final implementation doesn't greatly improve the original design − we actually even worsen the coremark score.

Not all approaches were conclusives, either because they were too complex for us to implement or they did not deliver the performance boost that we expected. However, we speculate that our ideas were ill-suited to perform well in coremark. Indeed, this benchmark heavily emphasizes on the execution of mathematical operations whereas we essentially worked on improving data flow (prefetching data in case of large array reading, prefetching instructions in case of frequent jumps/branches).

In the end, despite our lack of strong results, we are quite satisfied with the amount of work we put in this project. We also learned a lot about designing and debugging CPUs and of course about RISC-V.

We thank the organizers for giving us the opportunity to work on such project and our teachers who gave us the knowledge and ressources necessary to participate in this contest.