2021 Past Paper Solutions

This is not an official answer and please take a grain of salt on this. Attached is a copy of exam feedback.

1. a. Under what circumstances can the XT-910 execute a conditional branch instruction with zero pipeline stalls?

    When the conditional branch prediction hits in the L0 BTB.

   b. When do you think instructions are allocated into the loop buffer (LBUF)?

    When the instructions represent a loop body and its total instruction count is less or equal to 16. Additionally, the loop body must be simple, especially not contain any inner loops. In other words, if there is a nested loop then the one allocated into the loop buffer should be the innermost loop.

   c. Why is the LBUF flushed on context switches?

    LBUF should be flushed during context switches because the code inside LBUF belongs to the old process which will not be used in the new process, and most importantly for the purpose of isolating memory content (the instructions in the LBUF) between different processes due to security reasons.

   d. What other events could trigger the processor to stop issuing from the LBUF, and use the L1 instruction cache instead? Identify (and explain briefly) two different events.

    There is a jump statement branching off to another location of the program so that it will fetch instruction from that location, which is in L1 instruction cache and thus the processor will stop issuing from the LBUF

    Another event that could trigger this transition is an exception. For example, if the loop body contains something like division by zero, then surely the rest of the instruction in LBUF will not be executed and the instructions to handle the exception will be executed. The instructions of exception handler will be brought from main memory to L1 instruction cache during IF stage.

   e. Some of the benefits of the loop buffer might be achievable in software. Give one argument for a software alternative (explaining briefly how it would work), and one argument in favour, instead, of the hardware loop buffer described in the article.

    A software implemention will not be limited by the hardware implementation of 16 entries/instructions. It can unroll as many instructions as needed.

    A possible software implementation to have the same effect as LBUF is loop unrolling: we can unroll the loop several times in order to reduce the number of jump instructions. For instance, unrolling 5 times will reduce the number of jumps to 1/5 of the original code.

    A hardware implementation can operate on the original instruction and thus the instruction count/binary file size could be smaller: software loop unrolling will introduce several times more instructions, which increase the size of the binary. Hardware implementaion also only has one jump instead of few: software loop unrolling can only reduce the number of jumps such that after executing the unrolling loops there will be a jump. In the hardware implementation, there is only 1 jump instruction at the end: essentially this is equivalent to unroll the loop infinite number of times.

    Hardware loop buffer also reduces the IFU power consumption as mentioned in the paper.

   f. The XT-910 executes one thread on each core at a time. Many competing designs, and GPUs, use hardware multi-threading (SMT or FGMT) to run two or more threads on the same core

    i) Which parts of the microarchitecture would have to be replicated, one per thread?

The program counter, the set of registers file including both the common registers and floating point/vector registers have to be replicated.

No need to replicate ALU/execution units, RUU, etc. since they are shared between hardware threads

ii) Would SMT introduce any opportunities for simplifying the Instruction Fetch Unit? Explain your answer.

Yes but not always. With SMT, the instructions are scheduled dynamically, which means that instructions from one thread might fill up the stalls/gaps when the other threads are stalled/waiting. Essentially, the instructions are contending for resources such as the execution units or RUU. However, when there is only one thread, this benefit will not appear and a complex IFU that deals stalls better would perform better.

iii) A colleague suggests that adding SMT would make the data prefetching mechanisms unnecessary. Are they right? Explain your answer.

It is true that SMT would hide more latency and saturate for memory access parallelism, e.g. accessing memory appears to be faster/more efficient because there is less stall due to high utilization of processor core in SMT. However, data prefetching will still produce better result since it allows more accesses on the flight.

2. a. The XT-910's ROB can hold up to 192 instructions. What happens when the ROB is full?

   When the ROB is full, there is no space left to retire a new instruction, thus the entire pipeline has to be stalled until the ROB is not full again.

   b. How does register renaming “eliminate the use of the costly move instructions”?

   Register renaming will assign commit-side register alias pointer to the same ROB entry pointed by issue-side register alias pointer. This has the same effect as “mov x0, x1” as x1 is basically an alias pointing to the same entry in the ROB as x0.

   c. What is the function of the “Age-Vector” scheduling algorithm?

   It helps to decide the priority of the instructions that are ready to be dispatched/issued. The instructions that watied the longest should be dispatched first, just like how priority-based process scheduling works in operating system.

   d. Physical registers are released when instructions are retired. What other event could result in a physical register being released?

   When there is a branch misprediction such that all the subsequent entries in the ROB should be flushed and register should be released. Similarly it will happen when there is a context switch

   e. What would go wrong if (due to a design bug) a physical register were accidentally not released?

   If a physical register is not released, then it will not be available later in the execution, which will reduce the number of register that will be available to use by the program. This might lead to a deadlock

   f. A context-switch occurs when execution transfers from one Linux process to another. List (with brief explanation) four architectural features in the XT-910 that may contribute to the overhead of context switches.

   Pipeline refill: the instructions of the new process needs to refill up the pipeline in order to meet the throughput

   Cached data and TLB content, as well as branch predictions will no longer be useful

Number of logic registers, including vector registers

g. On RISCV, a spinlock is implemented with a loop that uses a load-reserved instruction, followed by a store conditional instruction:

i) How is execution of such a spinlock affected by the arrival of a cacheline invalidation?

A cache invalidation will make the store conditional fail, which will cause the loop to run again and try to lock it again.

ii) When and where is it detected?

The cache invalidation is detected by the cache controller the multi-core interconnection network. When it tries to run the store conditional, it will first check with the interconnection network whether the cache copy on this core is valid. If so then execute and hold the lock; if not, then it fails and tries to acquire the lock again.

3. a. What is the purpose of the "Early forward" arrow in Figure 9?

Early forward is used in order to directly forward the data requested by the load instruction to the execution unit even before writing to the register, so that the processor can continue execution while the data is written in the WB stage. This causes execution to restart early and improves efficiency.

b. When the Retirement Unit encounters an instruction that results in an exception or misprediction, speculative instructions of following instructions must be flushed:

i) Under what circumstances might a speculative execution failure occur in the branch pipe (BR, Fig.4)?

When there are two branch instructions executed in the pipeline and the second one is just entering the branch pipe (BR) due to speculative execution, but the Retirement Unit just discovers that the first branch is not taken (but previously predicted as taken).

Or simply because there is an exception happens

ii) What happens when a load instruction attempts to access an invalid address?

When a load instruction attempts to load data from an address, L0 TLB is first checked. If there is a miss then L1 TLB is checked. If there is still a miss, then MMU will perform virtual address translation table walk to determine the physical address. If at any point of the translation process, it discovers that the given address is invalid, then the load instruction should be marked and only throws a fault when it is committed.

iii) Under what circumstances, apart from invalid addresses, might a speculative execution failure occur in the Load-Store Unit (LSU, Fig.9)?

When executing a store instruction, there is a load that appears after the store in the program order but gets to execute earlier in the LSU pipeline, just as described in the article.

c. According to the article, "By separating the address and the data stores, the address generation and cache access can be performed earlier" (line 722). Why is it useful to initiate cache access for a store instruction as soon as the store address is known, even if the store data is not yet ready?

As soon as the store address is known, we can perform address translation and relevant cacheline can be allocated into L1 cache, while we wait for the data to be stored. Assuming the cache policy is write-allocate (so that we have to bring the cacheline into L1 cache and write the data to L1 cache, not the main memory).

d. According to the article, the XT-910 “can broadcast TLB maintenance information through the interconnection bus.” (line 849). Under what circumstances might it be necessary to update the TLB of another core?

When, for instance, both core runs different threads of the same process, and one of them malloced/freed a variable. In this case, this thread needs to let the other thread running on the other core knows that the TLB entry has been created/invalidated. This is especially necessary when freeing a variable: the TLB on the other core really needs to know that this variable is freed and the corresponding physical address is no longer available.

e. What problem with cache coherency would be made worse if the cache line size were increased? Illustrate your answer with a simple example program fragment.

If the cache line size is increased, then since cache coherency information is stored per cache line, there will be more frequent invalidation passed between the cores. Below is an example of what the above explanation means:

```
int a[128]
...
// Below are two threads of the same process running on two cores

// same process running on core 1
a[31] = 0;

// same process running on core 2
a[127] = 0;
```

If the cache line size is 32 Bytes, then no cache invalidation will be passed around since `a[31]` and `a[127]` are essentially in different cachelines. However, if the cache line size is 128 Bytes, then they belong to the same cacheline such that wen core 1 writes `a[31]` , it has to tell core 2 that the same cachline in core 2 is invalidated. Same happens when core 2 writes `a[127]`, which results in many invalidation broadcasts.

f. We studied an implementation of the Spectre attack (variant 1), which defeats language-based security using bounds checking, in the lectures. Do you think it would work on the XT-910? Explain your answer carefully.

It is possible that it will work on XT-910 since it supports speculative execution. It also depends on whether XT-910 has implemented load access checking or prevention of fine-grained timing to avoid Spectre 1 attack.

Spectre 1 works by triggering speculative execution of load into L1 cache. Then the attacker can use timing to determine whether the cache line has been allocated as a result of the speculative load.

The overall average for this paper was on-target, but students were quite widely spread – reflecting that the exam was quite challenging for some students. Quite a few students lost marks through answering briefly, without thinking about how to show understanding – and, sometimes, without thinking about what the question was really looking for.

Q1 a: Good answers referred to the L0BTB,and commented that the later predictors would also have to agree in order to avoid a resteer.

Q1 b: This question required students to use some imagination as the details of the LBUF were not covered in the lectures, nor the XT910 paper. Students who answered by discussing at which pipeline stage LBUF allocation occurs got some credit. Better answers addressed the question of under what circumstances allocation might take place. Some loop buffer implementations always allocate, while some use a predictor of some kind to decide when to allocate.

Q1 d: The question here asked what events might lead the processor to *stop* issuing from the LBUF – so the assumption is that until that point it was issuing from the LBUF. Jumping out, simply exiting, or an interrupt or exception are all good answers.

Q1 e: Software pipelining, or simply unrolling, are good answers – the key thing is to avoid the loss of issue opportunities caused by the loop branch. Software solutions avoid the limitation on loop length. Good answers mentioned the advantage for the loop buffer of reducing instruction fetch power consumption.

Q1 f: Many students were surprisingly confused over what resources *have* to be replicated to support SMT or FGMT: PCs, register files (or at least register alias tables), register-address predictors *must* replicated. Most other resources might optionally be increased.

FGMT (where instructions are issued round-robin from each of the threads allocated to the core) should definitely reduce the IFU complexity, as it should avoid the complexity (described in the paper) of handling back-to-back branches within a single thread. Whether the same simplification is a good idea with SMT is less clear, as we might aim for one thread to be able to use the whole core when no other threads are present.

With enough SMT threads on a core, the number of memory accesses in flight may be enough to saturate the memory system's parallelism, and might be enough to hide quite a lot of access latency. The best answers commented that to really hide typical memory access latency, you would need a lot of threads. Hardware prefetching provides a way to get a lot more accesses in-flight at once, at lower cost.

Q2 b: Good answers here explained how for a register-register *"move a b"* the pointer for register *a* in the issue-side register alias table (RAT) can be copied to *b* at the IR stage, so that the physical register is then pointed to by both RAT entries. Weaker answers discussed register renaming generally without explaining this.

Q2 c: Good answers here referred to the problem of deciding priority when multiple instructions are ready to be dispatched – and that it's a good heuristic to dispatch instructions that have been waiting longest.

Q2 e: The right answer here is that if registers are not released, they can't be re-allocated – so the pool of available registers will get reduced. This could eventually lead to deadlock. Some students suggested that failing to release a register might lead to a wrong result, which seems unlikely.

Q2 f: Good answers here mentioned the number of (architectural) registers, including vector registers. Another good observation is that cached data, TLB entries and branch predictions will no longer be useful, so additional misses and mispredictions will result. Some students suggested that caches etc should be flushed - this would normally not be done on a process switch, as the costs would be immense. It is not necessary to flush the physical registers or ROB. Some students discussed the need to flush the TLB – however, as the XT910 article mentions, the use of ASIDs should make this very rare.

Q2 g: Some students simply discussed cache coherency – which is OK, but it's not the whole story. The key idea behind the load-reserved mechanism is that any invalidation of the target address has to cause the store-conditional operation to fail. It's OK to say this happens in the cache controller (the snoop filter is something else, but it's close). The subtlety is that the store-conditional must be committed iff the load-reserved target is not invalidated – so there is a need for a coupling between committing the store-conditional instruction and completing the resulting store.

Q3 b (ii): Clearly a load from an invalid address should result in a fault. The question was looking for how this is detected, and how this (might) lead to a flush. The address is presumably checked in the L0TLB. If it's a miss, we have to check the L1TLB, and eventually we may need to walk the page table. If, at some point, the address is found to be invalid, the ROB entry of the load needs to be marked. But the exception should only be thrown if the faulting instruction is committed.

Q3 c: A simple good answer here is to explain that a store usually requires the relevant cache line to be loaded and allocated into the L1 cache. This can be done without waiting for the value that will eventually actually be stored. Other good answers commented that this strategy allows address translation to occur in parallel with evaluating the store value.

Q3 e: Some students commented that using large cache lines might lead to more data being transferred when remote accesses occur. However this might actually be a good thing, if we have spatial locality. Good answers concerned examples where two processors try to write to the same cache line – "false sharing".

Q3 f: The article really does not comment on sidechannel vulnerabilities, so students were required to speculate. A common error was to refer to multiple threads, or context switches – but the question concerns Spectre Variant 1, which is about bounds checking within a single thread's address space. Good answers showed understanding of how the vulnerability works, and pointed to relevant mechanisms in the microarchitecture that appear to enable it. The best answers speculated on how, despite this, mitigation might be possible – perhaps by preventing fine-grain timing, or by checking a load's access validity before forwarding the loaded value.

Overall, most students displayed the results of a lot of hard work, and showed mature facility with complex architectural ideas. It was inspiring to see what you can do!