Module 5: "MIPS R10000: A Case Study"
Lecture 9: "MIPS R10000: A Case Study"

MIPS R10000

A case study in modern microarchitecture

- Overview
- Stage 1: Fetch
- Stage 2: Decode/Rename
- Branch prediction
- Branch predictor
- Register renaming
- Preparing to issue
- Stage 3: Issue
- Load-dependents
- Functional units
- Result writeback
- Retirement or commit

[Reference: K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. IEEE Micro, 16(2): 28-40, April 1996.]

Previous Next

Module 5: "MIPS R10000: A Case Study"

Lecture 9: "MIPS R10000: A Case Study"

Overview

- Mid 90s: One of the first dynamic out-of-order superscalar RISC microprocessors
- 6.8 M transistors on 298 mm2 die (0.35 µm CMOS)
- Out of 6.8 M transistors 4.4 M are devoted to L1 instruction and data caches
- Fetches, decodes, renames 4 instructions every cycle
- 64-bit registers: the data path width is 64 bits
- On-chip 32 KB L1 instruction and data caches, 2-way set associative
- Off-chip L2 cache of variable size (512 KB to 16 MB), 2-way set associative, line size 128 bytes

Stage 1: Fetch

- The instructions are slightly pre-decoded when the cache line is brought into Icache
  - Simplifies the decode stage
- Processor fetches four sequential instructions every cycle from the Icache
- The iTLB has eight entries, fully associative
- No BTB
- So the fetcher really cannot do anything about branches other than fetching sequentially

Stage 2: Decode/Rename

- Decodes and renames four instructions every cycle
- The targets of branches, unconditional jumps, and subroutine calls (named jump and link or jal) are computed in this stage
- Unconditional jumps are not fed into the pipeline and the fetcher PC is modified directly by the decoder
- Conditional branches look up a simple predictor to predict the branch direction (taken or not taken) and accordingly modify the fetch PC

Previous Next

Branch prediction

- Branches are predicted and unconditional jumps are computed in stage 2
  - There is always a one-cycle bubble (four instructions)
- In case of branch misprediction (which will be detected later) the processor may need to roll back and restart fetching from the correct target
  - Need to checkpoint (i.e. save) the register map right after the branch is renamed (will be needed to restore in case of misprediction)
- The processor supports at most four register map checkpoints; this is stored in a structure called branch stack (really, it is a FIFO queue, not a stack)
  - Can support up to four in-flight branches

Branch predictor

- The predictor is an array of 512 two-bit saturating counters
  - Can count up to 3; if already 3, an increment does not have any effect (remains at 3)
  - Similarly, if the count is 0, a decrement does not have any effect (remains at 0)
- The array is indexed by PC[11:3]
  - Ignore lower 3 bits, take the next 9 bits
  - The outcome is the count at that index of the predictor
- If count >= 2 then predict taken; else not taken
- Very simple algorithm; prediction accuracy of 85+% on most benchmarks; works fine for short pipes
- Commonly known as bimodal branch predictor
- The branch predictor is updated when a conditional branch retires (in-order update because retirement is in-order)
  - At retirement we know the correct outcome of the branch
  - So we use that to train the predictor
  - If the branch is taken the count in the index for that branch is incremented (remains at 3 if already 3)
  - If the branch is not taken the count is decremented (remains at zero if already 0)
- This predictor will fail to predict many simple patterns including alternating branches depending on where the count starts

Register renaming

- Takes place in the second pipeline stage
- As we have discussed, every destination is assigned a new physical register from the free list
- The sources are assigned the existing map
- Map table is updated with the newly renamed dest.
- For every destination physical register, a busy bit is set high to signify that the value in this register is not yet ready; this bit is cleared after the instruction completes execution
- The integer and floating-point instructions are assigned registers from two separate free lists
  - The integer and fp register files are separate (each has 64 registers)

Previous Next

Preparing to issue

- Finally, during the second stage every instruction is assigned an active list entry
  - The active list is a 32-entry FIFO queue which keeps track of all in-flight instructions (at most 32) in-order
  - Each entry contains various info about the allocated instruction such as physical dest reg number etc.
- Also, each instruction is assigned to one of the three issue queues depending on its type
  - Integer queue: holds integer ALU instructions
  - Floating-point queue: holds FPU instructions
  - Address queue: holds the memory operations
- Therefore, stage 2 may stall if the processor runs out of: active list entries, physical regs, issue queue entries

Stage 3: Issue

- Three issue queue selection logics work in parallel
- Integer and fp queue issue logics are similar
- Integer issue logic
  - Integer queue contains 16 entries (can hold at most 16 instructions)
  - Search for ready-to-issue instructions among these 16
  - Issue at most two instructions to two ALUs
- Address queue
  - Slightly more complicated
  - When a load or a store is issued the address is still not known
  - To simplify matters, R10000 issues load/stores in-order (we have seen problems associated with out-of-order load/store issue)

Load-dependents

- The loads take two cycles to execute
  - During the first cycle the address is computed
  - During the second cycle the dTLB and data cache are accessed
  - Ideally I want to issue an instruction dependent on the load so that the instruction can pick up the load value from the bypass just in time
  - Assume that a load issues in cycle 0, computes address in cycle 1, and looks up cache in cycle 2
  - I want to issue the dependent in cycle 2 so that it can pick up the load value just before executing in cycle 3
  - Thus the load looks up cache in parallel with the issuing of the dependent; the dependent is issued even before it is known whether the load will hit in the cache; this is called load hit speculation (re-execute later if the load misses)

Previous Next

Functional units

- Right after an instruction is issued it reads the source operands (dictated by physical reg numbers) from the register file (integer or fp depending on instruction type)
- From stage 4 onwards the instructions execute
  - Two ALUs: branch and shift can execute on ALU1, multiply/divide can execute on ALU2, all other instructions can execute on any of the two ALUs; ALU1 is responsible for triggering rollback in case of branch misprediction (marks all instructions after the branch as squashed, restores the register map from correct branch stack entry, sets fetch PC to the correct target)
  - Four FPUs: one dedicated for fp multiply, one for fp divide, one for fp square root, most of the other instructions execute on the remaining FPU
  - LSU (Load/store unit): Address calc. ALU, dTLB is fully assoc. with 64 entries and translates 44-bit VA to 40-bit PA, PA is used to match dcache tags (virtually indexed physically tagged)

Result writeback

- As soon as an instruction completes execution the result is written back to the destination physical register
  - No need to wait till retirement since the renamer has guaranteed that this physical destination is associated with a unique instruction in the pipeline
- Also the results are launched on the bypass network (from outputs of ALU/FPU/dcache to inputs of ALU/FPU/address calculation ALUs)
  - This guarantees that dependents can be issued back-to-back and still they can receive the correct value
  - add r3, r4, r5; add r6, r4, r3; (can be issued in consecutive cycles, although the second add will read a wrong value of r3 from the register file)

Retirement or commit

- Immediately after the instructions finish execution, they may not be able to leave the pipe
  - In-order retirement is necessary for precise exception
- When an instruction comes to the head of the active list it can retire
- R10k retires 4 instructions every cycle
- Retirement involves
  - Updating the branch predictor and freeing its branch stack entry if it is a branch instruction
  - Moving the store value from the speculative store buffer entry to the L1 data cache if it is a store instruction
  - Freeing old destination physical register and updating the register free list
  - And, finally, freeing the active list entry itself

Previous Next

Self-assessment Exercise

These problems should be tried after module 05 is completed.

1. Consider the following memory organization of a processor. The virtual address is 40 bits, the physical address is 32 bits, the page size is 8 KB. The processor has a 4-way set associative 128-entry TLB i.e. each way has 32 sets. Each page table entry is 32 bits in size. The processor also has a 2-way set associative 32 KB L1 cache with line size of 64 bytes.

(A) What is the total size of the page table?
(B) Clearly show (with the help of a diagram) the addressing scheme if the cache is virtually indexed and physically tagged. Your diagram should show the width of TLB and cache tags.
(C) If the cache was physically indexed and physically tagged, what part of the addressing scheme would change?

2. A set associative cache has longer hit time than an equally sized direct-mapped cache. Why?

3. The Alpha 21264 has a virtually indexed virtually tagged instruction cache. Do you see any security/protection issues with this? If yes, explain and offer a solution. How would you maintain correctness of such a cache in a multi-programmed environment?

4. Consider the following segment of C code for adding the elements in each column of an NxN matrix A and putting it in a vector x of size N.

```
for(j=0;j<N;j++) {
for(i=0;i<N;i++) {
x[j] += A[i][j];
}
}
```

Assume that the C compiler carries out a row-major layout of matrix A i.e. A[i][j] and A[i][j+1] are adjacent to each other in memory for all i and j in the legal range and A[i][N-1] and A[i+1][0] are adjacent to each other for all i in the legal range. Assume further that each element of A and x is a floating point double i.e. 8 bytes in size. This code is executed on a modern speculative out-of-order processor with the following memory hierarchy: page size 4 KB, fully associative 128-entry data TLB, 32 KB 2-way set associative single level data cache with 32 bytes line size, 256 MB DRAM. You may assume that the cache is virtually indexed and physically tagged, although this information is not needed to answer this question. For N=8192, compute the following (please show all the intermediate steps). Assume that every instruction hits in the instruction cache. Assume LRU replacement policy for physical page frames, TLB entries, and cache sets.

(A) Number of page faults.

(B) Number of data TLB misses.
(C) Number of data cache misses. Assume that x and A do not conflict with each other in the cache.
(D) At most how many memory operations can the processor overlap before coming to a halt? Assume that the instruction selection logic (associated with the issue unit) gives priority to older instructions over younger instructions if both are ready to issue in a cycle.

5. Suppose you are running a program on two machines, both having a single level of cache hierarchy (i.e. only L1 caches). In one machine the cache is virtually indexed and physically tagged while in the other it is physically indexed and physically tagged. Will there be any difference in cache miss rates when the program is run on these two machines?

Previous Next

Solution of Self-assessment Exercise

1. Consider the following memory organization of a processor. The virtual address is 40 bits, the physical address is 32 bits, the page size is 8 KB. The processor has a 4-way set associative 128-entry TLB i.e. each way has 32 sets. Each page table entry is 32 bits in size. The processor also has a 2-way set associative 32 KB L1 cache with line size of 64 bytes.

(A) What is the total size of the page table?

Solution: The physical memory is 2^32 bytes, since the physical address is 32 bits. Since the page size is 8 KB, the number of pages is (2^32)/(2^13) i.e., 2^19. Since each page table entry is four bytes in size and each page must have one page table entry, the size of the page table is (2^19)*4 bytes or 2 MB.

(B) Clearly show (with the help of a diagram) the addressing scheme if the cache is virtually indexed and physically tagged. Your diagram should show the width of TLB and cache tags.

Solution: I will describe the addressing scheme here. You can derive the diagram from that. The processor generates 40-bit virtual addresses for memory operations. This address must be translated to a physical address that can be used to look up the memory (through the caches). The first step in this translation is TLB lookup. Since the TLB has 32 sets, the index width for TLB lookup is five bits. The lowest 13 bits of the virtual address constitute the page offset and are not used for TLB lookup. The next lower five bits are used for indexing into the TLB. This leaves the upper 22 bits of the virtual address to be used as the TLB tag. On a TLB hit, the TLB entry provides the necessary page table entry, which is 32 bits in width. On a TLB miss, the page table entry must be read from the page table resident in memory or cache. Nonetheless, the net effect of whichever path is taken is that we have the 32-bit page table entry. From these 32 bits, the necessary 19-bit physical page frame number is extracted (recall that the number of physical pages is 2^19). When the 13-bit page offset is concatenated to this 19-bit frame number, we get the target physical address. We must first look up the cache to check if the data corresponding to this address is already resident there before querying the memory. Since the cache is virtually indexed and physically tagged, the cache lookup can start at the same time as TLB lookup. The cache has (2^15)/(64*2) or 256 sets. So eight bits are needed to index the cache. The lower six bits of the virtual address are the block offset and not used for cache indexing. The next eight bits are used as cache index. The tags resident at both the ways of this set are read out. The target tag is computed from the physical address and must be compared against both the read out tags to test for a cache hit. Let's try to understand how the target tag is computed from the physical address that we have formed above with the help of the page table entry. Usually, the tag is derived by removing the block offset and cache index bits from the physical address. So, in this case, it is tempting to take the upper 18 bits of the physical address as the cache tag. Unfortunately, this does not work for virtually indexed

physically tagged cache where the page offset is smaller than the block offset plus cache index. In this particular example, they differ by one bit. Let's see what the problem is. Consider a two different cache blocks residing at the same cache index v derived from the virtual address. This means that these blocks have identical lower 14 bits of the virtual address. This guarantees that these two blocks will have identical lower 13 bits of physical address because virtual to physical address translation does not change page offset bits. However, nothing stops these two blocks from having identical upper 18 bits of the physical address, but different 14th bit. Now, it is clear why the traditional tag computation would make mistakes in identifying the correct block. So the cache tag must also include the 14th bit. In other words, the cache tag needs to be identical to the physical page frame number. This completes the cache lookup. On a cache miss, the 32-bit physical address must be sent to memory for satisfying the cache miss.

(C) If the cache was physically indexed and physically tagged, what part of the addressing scheme would change?

Solution: Almost everything remains unchanged, except that the cache index comes from the physical address now. As a result, the cache lookup cannot start until the TLB lookup completes. The cache tag now can be only upper 18 bits of the physical address.

2. A set associative cache has longer hit time than an equally sized direct-mapped cache. Why?

Solution: Iso-capacity direct-mapped cache has wider index decoder than a set associative cache. So index decoding takes longer in the direct-mapped cache. However, in set associative cache, a multiplexing stage is needed to choose from the possible candidates within the target set based on tag comparison outcome. While the decoder width falls logarithmically with set-associativity, the multiplexer width grows linearly. For example, a k-way set associative cache would require log(k) less index bits compared to an iso-capacity direct-mapped cache, but the multiplexing stage of the set associative cache would require a k-to-1 multiplexer. Overall, the multiplexer delay outweighs the gain in the decoder delay.

3. The Alpha 21264 has a virtually indexed virtually tagged instruction cache. Do you see any security/protection issues with this? If yes, explain and offer a solution. How would you maintain correctness of such a cache in a multi-programmed environment?

Solution: The main purpose of having a virtually indexed virtually tagged instruction cache is to get rid of the TLB from the instruction lookup path. However, this also removes the much-needed protection provided by the TLB entries. For example, now buggy codes can easily overwrite the instructions in the instruction cache. The minimal solution to this problem would still retain the read-write-execute permission bits in a TLB-like structure. This is essentially a translation-less TLB. In a multi-programmed environment, it becomes difficult to distinguish codes belonging to different processes in a virtually indexed virtually tagged cache because every process has the same virtual address map. Two possible solutions exist. On a context switch, one can flush the entire instruction cache. This may slightly elongate the context switch time and the process that is

switching in will see the cold start effect. Another solution would incorporate process id in the cache tag. However, this may increase the cache latency depending on the width of the process id. In general, it is very difficult to say which one is going to be better and depends on the class of applications that will run.

4. Consider the following segment of C code for adding the elements in each column of an NxN matrix A and putting it in a vector x of size N.

```
for(j=0;j<N;j++) {
for(i=0;i<N;i++) {
x[j] += A[i][j];
}
}
```

Assume that the C compiler carries out a row-major layout of matrix A i.e., A[i][j] and A[i][j+1] are adjacent to each other in memory for all i and j in the legal range and A[i][N-1] and A[i+1][0] are adjacent to each other for all i in the legal range. Assume further that each element of A and x is a floating point double i.e., 8 bytes in size. This code is executed on a modern speculative out-of-order issue processor with the following memory hierarchy: page size 4 KB, fully associative 128-entry data TLB, 32 KB 2-way set associative single level data cache with 32 bytes line size, 256 MB DRAM. You may assume that the cache is virtually indexed and physically tagged, although this information is not needed to answer this question. For N=8192, compute the following (please show all the intermediate steps). Assume that every instruction hits in the instruction cache. Assume LRU replacement policy for physical page frames, TLB entries, and cache sets.

(A) Number of page faults.

Solution: The total size of x is 64 KB and the total size of A is 512 MB. So, these do not fit in the physical memory, which is of size 256 MB. Also, we note that one row of A is of size 64 KB. As a result, every row of A starts on a new page. As the computation starts, the first outer loop iteration suffers from one page fault due to x and 8192 page faults due to A. Since one page can hold 512 elements of x and A, the next 511 outer loop iterations do not take any page faults. The j=512 iteration again suffers from one page fault in x and 8192 fresh page faults in A. This pattern continues until the memory gets filled up. At this point we need to invoke the replacement policy, which is LRU. As a result, the old pages of x and A will get replaced to make room for the new ones. Instead of calculating the exact iteration point where the memory gets exhausted, we only note that the page fault pattern continues to hold even beyond this point. Therefore, the total number of page faults is 8193*(8192/512) or 8193*16 or 131088.

(B) Number of data TLB misses.

Solution: The TLB can hold 128 pages at a time. The TLB gets filled up at j=0, i=126 with one translation for x[0] and 127 translations for A[0][0] to A[0][126]. At this point, the LRU replacement policy is invoked and it replaces the translations of A. The translation of x[0] does not get replaced because it is touched in every inner loop iteration. By the time the j=0 iteration is finished, only the last 127

translations of A survive in the TLB. As a result, every access of A suffers from a TLB miss because A is never able to reuse TLB translations because the reuse distance exceeds the TLB reach. On the other hand, x enjoys maximum possible reuse in the TLB. Therefore, every page of x suffers from exactly one TLB miss, while every access of A suffers from a TLB miss. So, the total number of TLB misses is 16+8192*8192 or 16+64M.

(C) Number of data cache misses. Assume that x and A do not conflict with each other in the cache.

Solution: In this case also, x enjoys maximum reuse, while A suffers from a cache miss on every access. This is because the number of blocks in the cache is 1024, which is much smaller than the reuse distance in A. One cache block can hold four elements of x. As a result, x takes a cache miss on every fourth element. So, the total number of cache misses is 2048+8192*8192 or 2K+64M.

(D) At most how many memory operations can the processor overlap before coming to a halt? Assume that the instruction selection logic (associated with the issue unit) gives priority to older instructions over younger instructions if both are ready to issue in a cycle.

Solution: Since every access of A suffers from a TLB miss and the TLB misses are usually implemented as restartable exceptions, there cannot be any overlap among multiple memory operations. A typical iteration would involve load of x, TLB miss followed by load of A, addition, and store to x. No two memory operations can overlap because the middle one always suffers from a TLB miss leading to a pipe flush.

5. Suppose you are running a program on two machines, both having a single level of cache hierarchy (i.e. only L1 caches). In one machine the cache is virtually indexed and physically tagged while in the other it is physically indexed and physically tagged. Will there be any difference in cache miss rates when the program is run on these two machines?

Solution: Depending on how the application is written, the virtually indexed cache may exhibit a higher miss rate only if the cache organization is such that the block offset plus the index bits exceed the page offset. We have seen above that in such situations the tag of the virtually indexed cache has to be extended to match the physical page frame number. As a result, the number of different cache blocks that can map to a cache index is larger in a virtually indexed cache. This increases the chance of conflict misses.

Previous Next