CSE 502: Computer Architecture

Instruction Commit
The End of the Road (um... Pipe)

• Commit is typically the last stage of the pipeline

• Anything an insn. does at this point is *irrevocable*
  
  – Only actions following sequential execution allowed
  
  – E.g., wrong path instructions may not commit
    
    • They do not exist in the sequential execution
Everything Should Appear In-Order

- ISA defines program execution in sequential order
- To the outside, CPU must appear to execute in order
“Looking” at CPU State

• When OS swaps contexts
  – OS saves the current program state (requires “looking”)
  – Allows restoring the state later

• When program has a fault (e.g., page fault)
  – OS steps in and “looks” at the “current” CPU state
Implementation in the CPU

• ARF keeps state corresponding to committed insns.
  – Commit from ROB happens in order
  – ARF always contains some RF state of sequential execution

• Whoever wants to “look” should look in ARF
  – What about insns. that executed out of order?
Only the Sequential Part Matters

Sequential View of the Processor

- Memory
- RF
- PC

State of the Superscalar Out-of-Order Processor

- Memory
- LSQ
- ROB
- RS
- PRF
- RF
- fPC
- PC

What if there’s no ARF?
If you need to “see” a register, you go through the aRAT first.
Wrong-path instructions are flushed…
architected state has never been touched

Fetch correct path instructions

Which can update the architected state when they commit
Committing Instructions (1/2)

- “Retire” vs. “Commit”
  - Sometimes people use this to mean the same thing
  - Sometimes they mean different things
    - Check the context!

- Insn. **commits** by making “effects” visible
  - (A)RF, Memory/$, PC
Committing Instructions (2/2)

• When an insn. executes, it modifies processor state
  – Update a register
  – Update memory
  – Update the PC

• To make “effects” visible, core copies values
  – Value from Physical Reg to Architected Reg
  – Value from LSQ to memory/cache
  – Value from ROB to Architected PC
x86 Commit (1/2)

- ROB contains uops, outside world knows insns.

If we take an interrupt right now, we’ll see a half-executed instruction!
x86 Commit (2/2)

• Works when uop-flow length ≤ commit width
• What to do with long flows?
  – In all cases: can’t commit until all uops in a flow completed
  – Just commit N uops per cycle
    ... but make commit uninterruptable

ROB

commit

Timer interrupt!

Defer: Can’t act on this yet...

Now do something about the interrupt.
Handling REP-prefixed Instructions (1/2)

- Ex. REP STOSB (memset EAX value ECX times)
  - Entire sequence is one x86 instruction
  - What if REPs for 1,000,000,000 iterations?
    - Can’t “lock up” for a billion cycles while uops commit
    - Can’t wait to commit until all uops are done
      - Can’t even fetch the entire instruction – not enough space in ROB

- At the ISA level, REP iterations are interruptible...
  - Treat each iteration as a separate “macro-op”
Handling REP-prefixed Instructions (2/2)

All of these are interruptible points (commit can stop and effects be seen by outside world), since they all have well-defined ISA-level states:

A: ECX=3, EDI = ptr+1
B: ECX=2, EDI = ptr+2
C: ECX=1, EDI = ptr+3
D: ECX=0, EDI = ptr+4

Check for zero iterations (could happen with MOV ECX, 0)

MOV EDI, <pointer>
SUB EAX, EAX
CLD
MOV ECX, 4
REP STOSB
ADD EBX, EDX

MOV EDI, xxx
SUB EAX, EAX
CLD
MOV ECX, 4
uCMP ECX, 0
uJCZ
STA tmp, EDI
STD EAX, tmp
ADD EDI, 1
SUB ECX, 1
uCMP ECX, 0
uJCZ
ADD EDI, 1
SUB ECX, 1
uCMP ECX, 0
uJCZ
ADD EBX, EDX

REP overhead for 1st iteration

REP overhead for 2nd iter.

REP overhead for 3rd iter

REP 4th iter

MOVS flow

MOVS flow

MOVS flow
Blocked Commit

- To commit N insns. per cycle, ROB needs N ports
  - (in *addition to* ports for dispatch, issue, exec, and WB)

Reduces cost, lowers IPC due to constraints.
Faults

• Divide-by-Zero, Overflow, Page-Fault
• All occur at a specific point in execution (precise)

CPU maintains appearance of sequential execution
Timing of DBZ Fault

- Need to hold on to your faults

On a fault, flush the machine and switch to the kernel

Let earlier instructions commit
- The arch. state is the same as just before the divide executed in the sequential order
- Now, raise the DBZ fault and when you switch to the kernel, everything appears as it should

Just make note of the fault, but don’t do anything (yet)
Speculative Faults

- Faults might not be faults...

Which is what we want, since in a sequential execution, the wrong-path divide would not have executed (and faulted)

Buffer faults until commit to avoid speculative faults
Timing of TLB Miss

- Store must re-execute (or re-commit)
  - Cannot leave the ROB

Store TLB miss can stall the core
Load Faults are Similar

• Load issues, misses in TLB
  – When load is oldest, switch to kernel for page-table walk
    ...could be painful; there are lots of loads

• Modern processors use hardware page-table walkers
  – OS loads a few registers with PT information (pointers)
  – Simple logic fetches mapping info from memory
  – Requires page-table format is specified by the ISA
Asynchronous Interrupts

• Some interrupts are not associated with insns.
  – Timer interrupt
  – I/O interrupt (disk, network, etc...)
  – Low battery, UPS shutdown

• When the CPU “notices” doesn’t matter (too much)
Two Options for Handling Async Interrupts

• Handle immediately
  – Use current architected state and flush the pipeline

• Deferred
  – Stop fetching, let processor drain, then switch to handler
    • What if CPU takes a fault in the mean time?
    • Which came “first”, the async. interrupt or the fault?
Store Retirement (1/2)

- Stores forward to later loads (for same address)
  - Normally, LSQ provides this facility

- Diagrams showing the forward store to later loads and updates to the cache.
Store Retirement (2/2)

• Can’t free LSQ Store entry until write is done
  – Enables forwarding until loads can get value from cache
• Have to re-check TLB when doing write
  – TLB contents at Execute were speculative
• Store may stall commit for a long time
  – If there’s a cache miss
  – If there’s a TLB miss (with HW TLB walk)

Don’t we check DTLB during store-address computation anyway?
Do we need to do it again here?
All instructions may have successfully executed, but none can commit!
Writeback Buffer (1/2)

- Want to get stores out of the way quickly

Eventually, the cache update occurs, the WB buffer entry is emptied.

Even if store misses in cache, entering WB buffer counts as committing.

Allows other insns. to commit.

WB buffer is part of the cache hierarchy. May need to provide values to later loads.

Cache can now provide the correct value.

Usually fast, but potential structural hazard
Writeback Buffer (2/2)

- Stores enter WB Buffer in program order
- Multiple stores can exist to the same address
  - Only the last store is “visible”

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>1234</td>
</tr>
<tr>
<td>13</td>
<td>-1</td>
</tr>
<tr>
<td>8</td>
<td>90901</td>
</tr>
<tr>
<td>42</td>
<td>5678</td>
</tr>
</tbody>
</table>

No one can “see” this store anymore!
Write Combining Buffer (1/2)

- Augment WBB to combine writes together

If Stores to same address, combine the writes
Write Combining Buffer (2/2)

• Can combine stores to same cache line

<table>
<thead>
<tr>
<th>$-Line Addr</th>
<th>Cache Line Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>1234</td>
</tr>
<tr>
<td></td>
<td>5678</td>
</tr>
</tbody>
</table>

One cache write can serve multiple original store instructions

Aggressiveness of write-combining may be limited by memory ordering model

Benefit: reduces cache traffic, reduces pressure on store buffers

Writeback/combining buffer can be implemented in/integrated with the MSHRs

Only certain memory regions may be “write-combinable” (e.g., USWC in x86)
Senior Store Queue

- Use STQ as WBB (not necessarily write combining)

While stores are completing, other accesses (loads, etc…) can continue getting the values from the “senior” STQ

New stores cannot allocate into Senior STQ entries until stores complete

No WBB and no stall on Store commit
Retire

• Insn. **retires** by cleaning up all related state
• Besides updating architected state
  ... needs to deallocate resources
  – ROB/LSQ entries
  – Physical register
  – “colors” of various sorts
  – RAT checkpoints
• Most are FIFO’s or Queues
  – Alloc/dealloc is usually just inc/dec head/tail pointers
• Unified PRF requires a little more work
  – Have to return “old” mapping to free list