Lecture: Out-of-order Processors

- Topics: load-store queues, memory dependences, SMT, caching intro
Out-of-Order Loads/Stores

What if the issue queue also had load/store instructions? Can we continue executing instructions out-of-order?
Memory Dependence Checking

- The issue queue checks for *register dependences* and executes instructions as soon as registers are ready.
- Loads/stores access memory as well – must check for RAW, WAW, and WAR hazards for memory as well.
- Hence, first check for register dependences to compute effective addresses; then check for memory dependences.
Memory Dependence Checking

<table>
<thead>
<tr>
<th>Ld</th>
<th>0x abcdef</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld</td>
<td></td>
</tr>
<tr>
<td>St</td>
<td></td>
</tr>
<tr>
<td>Ld</td>
<td></td>
</tr>
<tr>
<td>Ld</td>
<td>0x abcdef</td>
</tr>
<tr>
<td>St</td>
<td>0x abcd00</td>
</tr>
<tr>
<td>Ld</td>
<td>0x abc000</td>
</tr>
<tr>
<td>Ld</td>
<td>0x abcd00</td>
</tr>
</tbody>
</table>

- Load and store addresses are maintained in program order in the Load/Store Queue (LSQ)
- Loads can issue if they are guaranteed to not have true dependences with earlier stores
- Stores can issue only if we are ready to modify memory (can not recover if an earlier instr raises an exception) – happens at commit
The Alpha 21264 Out-of-Order Implementation

Branch prediction and instr fetch

R1 ← R1+R2
R2 ← R1+R3
BEQZ R2
R3 ← R1+R2
R1 ← R3+R2
LD R4 ← 8[R3]
ST R4 → 8[R1]

Instr Fetch Queue

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
Instr 6
Instr 7

Decode & Rename

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
Instr 6
Instr 7

Reorder Buffer (ROB)

P33 ← P1+P2
P34 ← P33+P3
BEQZ P34
P35 ← P33+P34
P36 ← P35+P34
P37 ← 8[P35]
P37 → 8[P36]

Register File P1-P64

ALU
ALU
ALU

Results written to regfile and tags broadcast to IQ

ALU

D-Cache

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
Instr 6
Instr 7

Committed Reg Map
R1→P1
R2→P2

Speculative Reg Map
R1→P36
R2→P34

Issue Queue (IQ)

P37 ← [P35 + 8]
P37 → [P36 + 8]

LSQ

Results written to regfile and tags broadcast to IQ

ALU

D-Cache
Problem 2

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume no memory dependence prediction.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>R1</td>
<td>[R2]</td>
<td>3</td>
<td>abcd</td>
</tr>
<tr>
<td>LD</td>
<td>R3</td>
<td>[R4]</td>
<td>6</td>
<td>adde</td>
</tr>
<tr>
<td>ST</td>
<td>R5</td>
<td>[R6]</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>LD</td>
<td>R7</td>
<td>[R8]</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>R9</td>
<td>[R10]</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>LD</td>
<td>R11</td>
<td>[R12]</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Problem 2

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume no memory dependence prediction.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>R1 ← [R2]</td>
<td>3</td>
<td>abcd</td>
<td>4</td>
</tr>
<tr>
<td>LD</td>
<td>R3 ← [R4]</td>
<td>6</td>
<td>adde</td>
<td>7</td>
</tr>
<tr>
<td>ST</td>
<td>R5 → [R6]</td>
<td>4</td>
<td>7</td>
<td>abba</td>
</tr>
<tr>
<td>LD</td>
<td>R7 ← [R8]</td>
<td>2</td>
<td>abce</td>
<td>3</td>
</tr>
<tr>
<td>ST</td>
<td>R9 → [R10]</td>
<td>8</td>
<td>3</td>
<td>abba</td>
</tr>
<tr>
<td>LD</td>
<td>R11 ← [R12]</td>
<td>1</td>
<td>abba</td>
<td>2</td>
</tr>
</tbody>
</table>
Problem 3

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume no memory dependence prediction.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>R1</td>
<td></td>
<td></td>
<td>abc</td>
</tr>
<tr>
<td>LD</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>R5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>R7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>R9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>R11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD  R1 ← [R2]  3  abc
LD  R3 ← [R4]  6  adde
ST  R5 → [R6]  5  7  abba
LD  R7 ← [R8]  2  abce
ST  R9 → [R10] 1  4  abba
LD  R11 ← [R12] 2  abba
Problem 3

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume no memory dependence prediction.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>R1 ← [R2]</td>
<td>3</td>
<td>abcd</td>
<td>4</td>
</tr>
<tr>
<td>LD</td>
<td>R3 ← [R4]</td>
<td>6</td>
<td>adde</td>
<td>7</td>
</tr>
<tr>
<td>ST</td>
<td>R5 → [R6]</td>
<td>5</td>
<td>7</td>
<td>abba</td>
</tr>
<tr>
<td>LD</td>
<td>R7 ← [R8]</td>
<td>2</td>
<td>abce</td>
<td>3</td>
</tr>
<tr>
<td>ST</td>
<td>R9 → [R10]</td>
<td>1</td>
<td>4</td>
<td>abba</td>
</tr>
<tr>
<td>LD</td>
<td>R11 ← [R12]</td>
<td>2</td>
<td>abba</td>
<td>3</td>
</tr>
</tbody>
</table>
Problem 4

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume memory dependence prediction.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>R1 ← [R2]</td>
<td>3</td>
<td>abcd</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>R3 ← [R4]</td>
<td>6</td>
<td>adde</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>R5 → [R6]</td>
<td>4 7</td>
<td>abba</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>R7 ← [R8]</td>
<td>2</td>
<td>abce</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>R9 → [R10]</td>
<td>8 3</td>
<td>abba</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>R11 ← [R12]</td>
<td>1</td>
<td>abba</td>
<td></td>
</tr>
</tbody>
</table>
Problem 4

Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. **Assume memory dependence prediction.**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1</td>
<td>[R2]</td>
<td>3</td>
<td>abcd</td>
<td>4</td>
</tr>
<tr>
<td>LD R3</td>
<td>[R4]</td>
<td>6</td>
<td>adde</td>
<td>7</td>
</tr>
<tr>
<td>ST R5</td>
<td>[R6]</td>
<td>4</td>
<td>7</td>
<td>abba</td>
</tr>
<tr>
<td>LD R7</td>
<td>[R8]</td>
<td>2</td>
<td>abce</td>
<td>3</td>
</tr>
<tr>
<td>ST R9</td>
<td>[R10]</td>
<td>8</td>
<td>3</td>
<td>abba</td>
</tr>
<tr>
<td>LD R11</td>
<td>[R12]</td>
<td>1</td>
<td>abba</td>
<td>2</td>
</tr>
</tbody>
</table>
Thread-Level Parallelism

• Motivation:
  ➢ a single thread leaves a processor under-utilized for most of the time
  ➢ by doubling processor area, single thread performance barely improves

• Strategies for thread-level parallelism:
  ➢ multiple threads share the same large processor \(\rightarrow\)
    reduces under-utilization, efficient resource allocation
    Simultaneous Multi-Threading (SMT)
  ➢ each thread executes on its own mini processor \(\rightarrow\)
    simple design, low interference between threads
    Chip Multi-Processing (CMP) or multi-core
How are Resources Shared?

Each box represents an issue slot for a functional unit. Peak thruput is 4 IPC.

- Superscalar processor has high under-utilization – not enough work every cycle, especially when there is a cache miss
- Fine-grained multithreading can only issue instructions from a single thread in a cycle – can not find max work every cycle, but cache misses can be tolerated
- Simultaneous multithreading can issue instructions from any thread every cycle – has the highest probability of finding work for every issue slot
What Resources are Shared?

• Multiple threads are simultaneously active (in other words, a new thread can start without a context switch)

• For correctness, each thread needs its own PC, IFQ, logical regs (and its own mappings from logical to phys regs)

• For performance, each thread could have its own ROB/LSQ (so that a stall in one thread does not stall commit in other threads), I-cache, branch predictor, D-cache, etc. (for low interference), although note that more sharing → better utilization of resources

• Each additional thread costs a PC, IFQ, rename tables, and ROB – cheap!
Pipeline Structure

Front End
Front End
Front End
Front End

Execution Engine

Private/Shared Front-end
Private Front-end

I-Cache
Bpred

Rename
ROB

Regs
IQ

DCache
FUs
Resource Sharing

Thread-1

- R1 ← R1 + R2
- R3 ← R1 + R4
- R5 ← R1 + R3

Instr Fetch

- P65 ← P1 + P2
- P66 ← P65 + P4
- P67 ← P65 + P66

Instr Rename

Thread-2

- R2 ← R1 + R2
- R5 ← R1 + R2
- R3 ← R5 + R3

Instr Fetch

- P76 ← P33 + P34
- P77 ← P33 + P76
- P78 ← P77 + P35

Instr Rename

Issue Queue

- P65 ← P1 + P2
- P66 ← P65 + P4
- P67 ← P65 + P66
- P76 ← P33 + P34
- P77 ← P33 + P76
- P78 ← P77 + P35

Register File

FU

FU

FU

FU

FU
Performance Implications of SMT

• Single thread performance is likely to go down (caches, branch predictors, registers, etc. are shared) – this effect can be mitigated by trying to prioritize one thread

• While fetching instructions, thread priority can dramatically influence total throughput – a widely accepted heuristic (ICOUNT): fetch such that each thread has an equal share of processor resources

• With eight threads in a processor with many resources, SMT yields throughput improvements of roughly 2-4
Multi-Programmed Speedup

- sixtrack and eon do not degrade their partners (small working sets?)
- swim and art degrade their partners (cache contention?)
- Best combination: swim & sixtrack
  worst combination: swim & art
- Static partitioning ensures low interference – worst slowdown is 0.9
**“ZEN 3” OVERVIEW**

**2 THREADS PER CORE (SMT)**

**STATE-OF-THE-ART BRANCH PREDICTOR**

**CACHES**
- I-cache 32k, 8-way
- Op-cache, 4K instructions
- D-cache 32k, 8-way
- L2 cache 512k, 8-way

**DECODE**
- 4 instructions / cycle decode or 8 ops from Op-cache
- 6 ops / cycle dispatched to Integer or FP

**EXECUTION CAPABILITIES**
- 4 integer units
- Dedicated branch and store data units
- 3 address generations per cycle
- 2 256-bit FP multiply accumulate / cycle

**3 MEMORY OPS PER CYCLE**

**TLBs**
- L1 64 entries I & D, all page sizes
- L2 512 I, 2K D, everything but 1G

---

**INSTRUCTION FETCH AND DECODE**

**INTEGER**
- 32K I-Cache 8 Way
- Decode
- Op Queue
- Dispatch
- 4 Instructions/Cycle

**FLOATING POINT**
- Branch Prediction
- Op-cache
- 8 Macro Ops/Cycle

---

**INTEGER**
- Integer Rename
- Scheduler
- Scheduler
- Scheduler
- Scheduler
- Integer Register File
- ALU BR
- ALU
- ACU
- ALU
- ACU
- ALU
- BR

**3 LOADS PER CYCLE**

**2 STORES PER CYCLE**

**FP Register File**
- Floating Point Rename
- Scheduler
- Scheduler
- F2I
- MUL MAC
- ADD
- MUL MAC
- ADD
AMD ZEN 3

**INT EXECUTION**

- New distributed scheduler organization
- Lower latencies for some instructions
- Larger out-of-order window
- 10 issue per cycle, up from 7

<table>
<thead>
<tr>
<th>RESOURCE</th>
<th>&quot;ZEN 2&quot;</th>
<th>&quot;ZEN 3&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer issue width</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>Integer register file</td>
<td>180</td>
<td>192</td>
</tr>
<tr>
<td>Integer scheduler</td>
<td>92</td>
<td>96</td>
</tr>
<tr>
<td>ROB</td>
<td>224</td>
<td>256</td>
</tr>
</tbody>
</table>

LOWER LATENCIES AND LARGER STRUCTURES TO EXTRACT ILP FOR FEEDING THE EXECUTION ENGINES

6 Macro Ops Per Cycle Dispatch

Integer Rename

Scheduler 3  Scheduler 2  Scheduler 1  Scheduler 0

Integer Register File

ALU3  BRU1  ALU2  AGU2  ALU1  AGU1  ALU0  BRU0  AGU0

Load/Store Unit

3X 64-bit loads
The Cache Hierarchy

Core → L1 → L2 → L3 → Off-chip memory
Problem 1

• Memory access time: Assume a program that has cache access times of 1-cyc (L1), 10-cyc (L2), 30-cyc (L3), and 300-cyc (memory), and MPKIs of 20 (L1), 10 (L2), and 5 (L3). Should you get rid of the L3?
Problem 1

- Memory access time: Assume a program that has cache access times of 1-cyc (L1), 10-cyc (L2), 30-cyc (L3), and 300-cyc (memory), and MPKIs of 20 (L1), 10 (L2), and 5 (L3). Should you get rid of the L3?

With L3: $1000 + 10 \times 20 + 30 \times 10 + 300 \times 5 = 3000$
Without L3: $1000 + 10 \times 20 + 10 \times 300 = 4200$