Multiple Instruction Issue/Virtual Memory Introduction

Guest Lecturer/TA: Andrew Luo

Great Idea #4: Parallelism

- Parallel Requests Assigned to computer e.g. search “Garcia”
- Parallel Threads Assigned to core e.g. lookup, ads
- Parallel Instructions > 1 instruction @ one time e.g. 5 pipelined instructions
- Parallel Data > 1 data item @ one time e.g. add of 4 pairs of words
- Hardware descriptions All gates functioning in parallel at same time

Review of Last Lecture (1/2)

A hazard is a situation that prevents the next instruction from executing in the next clock cycle
- Structural hazard: A required resource is needed by multiple instructions in different stages
- Data hazard: Data dependency between instructions
- Control hazard: The flow of execution depends on the previous instruction (ex. jumps, branches)

Review of Last Lecture (2/2)

Hazards hurt the performance of pipelined processors
- Stalling can be applied to any hazard, but hurt performance... better solutions?
- Structural hazards: Have separate structures for each stage
- Data hazards: Forwarding
- Control hazards: Branch prediction (mitigates), branch delay slot

Agenda

- Multiple Issue
- Administrivia
- Virtual Memory Introduction

Multiple Issue

- Modern processors can issue and execute multiple instructions per clock cycle
- CPI < 1 (superscalar), so can use Instructions Per Cycle (IPC) instead
- e.g. 4 GHz 4-way multiple-issue can execute 16 billion IPS with peak CPI = 0.25 and peak IPC = 4
- But dependencies and structural hazards reduce this in practice
Multiple Issue

- Static multiple issue
  - Compiler reorders independent/commutative instructions to be issued together
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines pipeline and chooses instructions to reorder/issue
  - CPU can resolve hazards at runtime

Pipeline Depth and Issue Width

Intel Processors over Time

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>3</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td>3</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core 2 Conroe</td>
<td>2006</td>
<td>2930 MHz</td>
<td>12-14</td>
<td>4*</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Pereney</td>
<td>2008</td>
<td>2930 MHz</td>
<td>12-14</td>
<td>4*</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Westmere</td>
<td>2010</td>
<td>3460 MHz</td>
<td>14</td>
<td>4*</td>
<td>6</td>
<td>130W</td>
</tr>
<tr>
<td>Xeon Sandy Bridge</td>
<td>2012</td>
<td>3100 MHz</td>
<td>14-19</td>
<td>4*</td>
<td>8</td>
<td>150W</td>
</tr>
<tr>
<td>Xeon Ivy Bridge</td>
<td>2014</td>
<td>2800 MHz</td>
<td>14-19</td>
<td>4*</td>
<td>15</td>
<td>155W</td>
</tr>
</tbody>
</table>

Static Multiple Issue

- Compiler reorders independent/commutative instructions to be issued together (an “issue packet”)
  - Group of instructions that can be issued on a single cycle
  - Determined by structural resources required
  - Specifies multiple concurrent operations

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies within a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with \texttt{nop}s if necessary
Dynamic Multiple Issue

- Used in “superscalar” processors
- CPU decides whether to issue 0, 1, 2, … instructions each cycle
  - Goal is to avoid structural and data hazards
- Avoids need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example:
  
  lw $t0, 20($s2)
  addu $t1, $t0, $t2
  subu $s4, $s4, $t3
  slti $t5, $s4, 20

  - Can start subu while addu is waiting for lw
- Especially useful on cache misses; can execute many instructions while waiting!

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g. cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined by I/O
- Different implementations of an ISA have different latencies and hazards
  - Forward compatibility and optimizations

Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right and roll back if necessary
- Examples:
  - Speculate on branch outcome (Branch Prediction)
    - Roll back if path taken is different
  - Speculate on load
    - Load into an internal register before instruction to minimize time waiting for memory
- Can be done in hardware or by compiler
  - Common to static and dynamic multiple issue

Not a Simple Linear Pipeline

3 major units operating in parallel:

- Instruction fetch and issue unit
  - Issues instructions in program order
- Many parallel functional (execution) units
  - Each unit has an input buffer called a Reservation Station
  - Holds operands and records the operation
  - Can execute instructions out-of-order (OOO)
- Commit unit
  - Saves results from functional units in Reorder Buffers
  - Stores results once branch resolved so OK to execute
  - Commits results in program order

Out-of-Order Execution (1/2)

Can also unroll loops in hardware

1) Fetch instructions in program order (≤ 4/clock)
2) Predict branches as taken/untaken
3) To avoid hazards on registers, rename registers using a set of internal registers (= 80 registers)
4) Collection of renamed instructions might execute in a window (= 60 instructions)
Out-of-Order Execution (2/2)

5) Execute instructions with ready operands in 1 of multiple functional units (ALUs, FPUs, Ld/St)
6) Buffer results of executed instructions until predicted branches are resolved in reorder buffer
7) If predicted branch correctly, commit results in program order
8) If predicted branch incorrectly, discard all dependent results and start with correct PC
Agenda
- Multiple Issue
- Administrivia
- Virtual Memory Introduction

Memory Hierarchy Requirements
- Principle of Locality
  - Allows caches to offer (close to) speed of cache memory with size of DRAM memory
  - Can we use this at the next level to give speed of DRAM memory with size of disk memory?
  - What other things do we need from our memory system?

Memory Hierarchy
- Regs
- Instr Operands
- L1 Cache
- Blocks
- L2 Cache
- Blocks
- Memory
- Pages
- Disk
- Files
- Tape
- Upper Level
- Faster
- Larger
- Lower Level

Administrivia
- HW5 due tonight
- Project 2 (Performance Optimization) due Sunday
- No lab today
  - TAs will be in lab to check off make up labs
  - Highly encouraged to make up labs today if you’re behind – treated as Tuesday checkoff for lateness
  - Project 3 (Pipelined Processor in Logisim) released Friday/Saturday

Memory Hierarchy Requirements
- Allow multiple processes to simultaneously occupy memory and provide protection
  - Don’t let programs read from or write to each other’s memories
- Give each program the illusion that it has its own private address space
  - Suppose a program has base address 0x00400000, then different processes each think their code resides at the same address
  - Each program must have a different view of memory
Virtual Memory

- Next level in the memory hierarchy
  - Provides illusion of very large main memory
  - Working set of “pages” residing in main memory
  (subset of all pages residing on disk)
- Main goal: Avoid reaching all the way back to disk as much as possible
- Additional goals:
  - Let OS share memory among many programs and protect them from each other
  - Each process thinks it has all the memory to itself

Virtual to Physical Address Translation

- Each program operates in its own virtual address space and thinks it’s the only program running
- Each is protected from the other
- OS can decide where each goes in memory
- Hardware gives virtual → physical mapping

VM Analogy (1/2)

- Trying to find a book in the UCB library system
  - Book title is like virtual address (VA)
    - What you want/are requesting
  - Book call number is like physical address (PA)
    - Where it is actually located
  - Card catalogue is like a page table (PT)
    - Maps from book title to call number
    - Does not contain the actual data you want
    - The catalogue itself takes up space in the library

VM Analogy (2/2)

- Indication of current location within the library system is like valid bit
  - Valid if in current library (main memory) vs. invalid if in another branch (disk)
  - Found on the card in the card catalogue

- Availability/terms of use like access rights
  - What you are allowed to do with the book
    (ability to check out, duration, etc.)
  - Also found on the card in the card catalogue

Mapping VM to PM

- Divide into equal sized chunks (usually 4-8 KiB)
- Any chunk of Virtual Memory can be assigned to any chunk of Physical Memory (“page”)

Paging Organization

- Here assume page size is 4 KiB
  - Page is both unit of mapping and unit of transfer between disk and physical memory

Physical Address

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x0000</td>
</tr>
<tr>
<td>0x0100</td>
<td>0x0100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x0700</td>
<td>0x0700</td>
</tr>
</tbody>
</table>

Virtual Memory

<table>
<thead>
<tr>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 MB</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Static</td>
</tr>
<tr>
<td>Static</td>
</tr>
<tr>
<td>PAGE 0</td>
</tr>
<tr>
<td>PAGE 1</td>
</tr>
<tr>
<td>PAGE 2</td>
</tr>
<tr>
<td>PAGE 31</td>
</tr>
</tbody>
</table>

Addr Trans MAP
Virtual Memory Mapping Function

- How large is main memory? Disk?
  - Don’t know! Designed to be interchangeable components
  - Need a system that works regardless of sizes
- Use lookup table (page table) to deal with arbitrary mapping
  - Index lookup table by # of pages in VM
    (not all entries will be used/valid)
  - Size of PM will affect size of stored translation

Address Mapping

- Pages are aligned in memory
  - Border address of each page has same lowest bits
- Page size [P bytes] is same in VM and PM, so denote lowest PO = \log_2(P) bits as page offset
- Use remaining upper address bits in mapping
  - Tells you which page you want (similar to Tag)

Address Mapping: Page Table

- Page Table functionality:
  - Incoming request is Virtual Address (VA), want Physical Address (PA)
  - Physical Offset = Virtual Offset (page-aligned)
  - So just swap Virtual Page Number (VPN) for Physical Page Number (PPN)
- Implementation?
  - Use VPN as index into PT
  - Store PPN and management bits (Valid, Access Rights)
  - Does NOT store actual data (the data sits in PM)

Page Table Layout

<table>
<thead>
<tr>
<th>Virtual Address:</th>
<th>VPN</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1) Index into PT usingVPN
2) Check Valid and Access Rights bits
3) Combine PPN and offset
4) Use PA to access memory

Page Table Entry Format

- Contains either PPN or indication not in main memory
  - Valid = Valid page table entry
    - \(1 \rightarrow\) virtual page is in physical memory
    - \(0 \rightarrow\) OS needs to fetch page from disk
  - Access Rights checked on every access to see if allowed (provides protection)
    - Read Only: Can read, but not write page
    - Read/Write: Read or write data on page
    - Executable: Can fetch instructions from page

Page Tables

- A page table (PT) contains the mapping of virtual addresses to physical addresses
- Where should PT be located?
  - Physical memory, so faster to access and can be shared by multiple processors
  - The OS maintains the PTs
    - Each process has its own page table
      - “State” of a process is PC, all registers, and PT
      - OS stores address of the PT of the current process in the Page Table Base Register
Caches vs. Virtual Memory

<table>
<thead>
<tr>
<th>Caches</th>
<th>Virtual Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>Page</td>
</tr>
<tr>
<td>Cache Miss</td>
<td>Page Fault</td>
</tr>
<tr>
<td>Block Size: 32-64B</td>
<td>Page Size: 4KiB-8KiB</td>
</tr>
<tr>
<td>Placement: Direct Mapped, N-way Set Associative</td>
<td>Fully Associative (almost always)</td>
</tr>
<tr>
<td>Replacement: LRU or Random</td>
<td>LRU</td>
</tr>
<tr>
<td>Write Thru or Back</td>
<td>Write Back</td>
</tr>
</tbody>
</table>

Notes on Page Table

- OS must reserve "Swap Space" on disk for each process
- To grow a process, ask Operating System
  - If unused pages, OS uses them first
  - If not, OS swaps some old pages to disk (Least Recently Used to pick pages to swap)
- Each process has own Page Table
- Will add details, but Page Table is essence of Virtual Memory

Technology Break

Notes on Page Table

Why would a process need to “grow”?

- A program's address space contains 4 regions:
  - stack: local variables, grows downward
  - heap: space requested for pointers via malloc(); resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change

Virtual Memory and Caches

- Physical memory is slow, so we cache data
  - Why not do the same with the page table?
- Translation Lookaside Buffer (TLB) is the equivalent of cache for the page table
Virtual to Physical Address Translation

1. Check TLB
2. Check page tables (in main memory)
3. Page fault – OS handles
4. OS: Check disk
5. OS: Swap on disk/Segmentation fault

Translation Look-AAside Buffers (TLBs)

- TLBs usually small, typically 128 - 256 entries
- Like any other cache, the TLB can be direct mapped, set associative, or fully associative

Context Switching and VM

- Context Switching now requires that both the TLB and caches be flushed
- In reality, TLB entries have a context tag

Peer Instruction

1) Locality is important yet different for cache and virtual memory (VM): temporal locality for caches but spatial locality for VM
2) VM helps both with security and cost

Peer Instruction Answer

1) No. Both for VM and cache
2) VM helps both with security and cost

Summary

- More aggressive performance options:
  - Longer pipelines
  - Superscalar (multiple issue)
  - Out-of-order execution
  - Speculation
- Virtual memory bridges memory and disk
  - Provides illusion of independent address spaces to processes and protects them from each other
  - VA to PA using Page Table
  - TLB