CS 61C: Great Ideas in Computer Architecture

Multiple Instruction Issue, Virtual Memory Introduction

Instructor: Justin Hsia
Great Idea #4: Parallelism

Software

- Parallel Requests
  Assigned to computer
  e.g. search “Garcia”

- Parallel Threads
  Assigned to core
  e.g. lookup, ads

Hardware

- Leverage Parallelism & Achieve High Performance

Hardware descriptions

- All gates functioning in parallel at same time

• 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

8/01/2013
Extended Review: Pipelining

• Why pipeline?
  – Increase clock speed by reducing critical path
  – Increase performance due to ILP

• How do we pipeline?
  – Add registers to delimit and pass information between pipeline stages

• Things that hurt pipeline performance
  – Filling and draining of pipeline
  – Unbalanced stages
  – Hazards
Pipelining and the Datapath

- Registers hold up information between stages
  - Each stage “starts” on same clock trigger
  - Each stage is working on a different instruction
  - Must pass ALL signals needed in any following stage
- Stages can still interact by going around registers
  - e.g. forwarding, hazard detection hardware
Pipelining Hazards (1/3)

• Structural, Data, and Control hazards
  – Structural hazards taken care of in MIPS
  – Data hazards only when register is written to and then accessed soon after (not necessarily immediately)

• Forwarding
  – Can forward from two different places (ALU, Mem)
  – Most further optimizations assume forwarding

• Delay slots and code reordering
  – Applies to both branches and jumps

• Branch prediction (can be dynamic)
Pipelining Hazards (2/3)

• **Identifying hazards:**
  1) Check system specs (forwarding? delayed branch/jump?)
  2) Check code for *potential* hazards
     • Find all loads, branches, and jumps
     • Anywhere a value is written to and then read in the next two instructions
  3) Draw out graphical pipeline and check if actually a hazard
     • When is result available? When is it needed?
Pipelining Hazards (3/3)

• **Solving hazards:**
  (assuming the following were not present already)
  – Forwarding?
    • Can remove 1 stall for loads, 2 stalls for everything else
  – Delayed branch?
    • Exposes delay slot for branch and jumps
  – Code reordering?
    • Find unrelated instruction from earlier to move into delay slot (branch, jump, or load)
  – Branch prediction?
    • Depends on exact code execution
Question: Which of the following signals is NOT needed to determine whether or not to forward for the following two instructions?

```
add  $s1, $t0, $t1
ori  $s2, $s1, 0xF
```

(A) rd.EX
(B) opcode.EX
(C) rt.ID
(D) opcode.ID

WHAT IF:
1) add was lw $s1,0($t0)?
2) random other instruction in-between?
3) ori was sw $s1,0($s0)?
Agenda

• Higher Level ILP
• Administrivia
• Dynamic Scheduling
• Virtual Memory Introduction
Higher Level ILP

1) Deeper pipeline (5 to 10 to 15 stages)
   – Less work per stage $\rightarrow$ shorter clock cycle

2) Multiple instruction issue
   – Replicate pipeline stages $\rightarrow$ multiple pipelines
   – Can start 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 reduce this in practice
Multiple Issue

• Static multiple issue
  – Compiler groups instructions to be issued together
  – Packages them into “issue slots”
  – Compiler detects and avoids hazards

• Dynamic multiple issue
  – CPU examines instruction stream and chooses instructions to issue each cycle
  – Compiler can help by reordering instructions
  – CPU resolves hazards using advanced techniques at runtime
Superscalar Laundry: Parallel per stage

- More resources, HW to match mix of parallel tasks?

8/01/2013 Summer 2013 -- Lecture #23
## 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>14</td>
<td>4</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>
Pipeline Depth and Issue Width

- Clock
- Power
- Pipeline Stages
- Issue width
- Cores
Static Multiple Issue

• Compiler groups instructions into *issue packets*
  – Group of instructions that can be issued on a single cycle
  – Determined by pipeline resources required
• Think of an issue packet as a very long instruction word (VLIW)
  – 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 *nops* if necessary
MIPS with Static Dual Issue

- Dual-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with \texttt{nop}

This prevents what kind of hazard?

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Datapath with Static Dual Issue

1 Extra 32-bit instruction

1 Extra Write Port
2 Extra Reg Read Ports

1 Extra Sign Ext

1 Extra ALU

8/01/2013  18Summer 2013 -- Lecture #23
Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet
    - \texttt{add $t0, $s0, $s1}
    - \texttt{load $s2, 0($t0)}
  - Splitting these into two packets is effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions/cycle
- More aggressive scheduling required!
Scheduling Example

• Schedule this for dual-issue MIPS

Loop:  lw  $t0, 0($s1)     # $t0=array element
      addu $t0, $t0, $s2  # add scalar in $s2
      sw  $t0, 0($s1)    # store result
      addi $s1, $s1,–4   # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0

<table>
<thead>
<tr>
<th>ALU/Branch</th>
<th>Load/Store</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>nop</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>lw  $t0, 0($s1)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>addi $s1, $s1,-4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>addu $t0, $t0, $s2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>bne  $s1, $zero, Loop</td>
<td>5</td>
</tr>
</tbody>
</table>

– IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

This change affects scheduling but not effect
Loop Unrolling

• Replicate loop body to expose more instruction level parallelism

• Use different registers per replication
  – Called *register renaming*
  – Avoid loop-carried *anti-dependencies*
    • Store followed by a load of the same register
    • a.k.a. “name dependence” (reuse of a register name)
Loop Unrolling Example

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: addi $s1, $s1, -16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>lw $t3, 4($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addu $t2, $t2, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>5</td>
</tr>
<tr>
<td>addu $t3, $t4, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>6</td>
</tr>
<tr>
<td>nop</td>
<td>sw $t2, 8($s1)</td>
<td>7</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td>8</td>
</tr>
</tbody>
</table>

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size
Agenda

• Higher Level ILP
• Administrivia
• Dynamic Scheduling
• Virtual Memory Introduction
Administrivia

• HW5 due tonight
• Project 2 Part 2 (and EC) due Sunday
• “Free” lab time today
  – The TAs will still be there, Lab 10 due Tue
• Project 3 will be posted by Friday night
  – Two-stage pipelined CPU in Logisim
Agenda

• Higher Level ILP
• Administrivia
• Dynamic Scheduling
• Virtual Memory Introduction
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 predicatable
  — e.g. cache misses
• Can’t always schedule around branches
  — Branch outcome is dynamically determined
• Different implementations of an ISA have different latencies and hazards
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
    • Roll back if location is updated

• Can be done in hardware or by compiler
• Common to static and dynamic multiple issue
Pipeline Hazard Analogy: Matching socks in later load

- A depends on D; stall since folder is tied up
A depends on D; let the rest continue
  – Need more resources to allow out-of-order (2 folders)
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)

Basically, **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)

Basically, **unroll loops in hardware**

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
Dynamically Scheduled CPU

Instruction fetch and decode unit

Reservation station

Reservation station

Reservation station

Reservation station

In-order issue

Wait here until all operands available

Out-of-order execute

Preserves dependencies

Commit unit

In-order commit

Can supply operands for issued instructions

Reorder buffer for register and memory writes

... and Hold

Execute...

Branch prediction, Register renaming

Functional units

Integer

Integer

Floating point

Load-store

8/01/2013 Summer 2013 -- Lecture #23
Out-Of-Order Intel

- All use O-O-O since 2001

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order/Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2006</td>
<td>2930MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>
AMD Opteron X4 Microarchitecture

Queues:
- 106 RISC ops
- 24 integer ops
- 36 FP/SSE ops
- 44 ld/st
AMD Opteron X4 Pipeline Flow

• For integer operations
  - 12 stages (Floating Point is 17 stages)
  - Up to 106 RISC-ops in progress

• Intel Nehalem is 16 stages for integer operations, details not revealed, but likely similar to above
  - Intel calls RISC operations “Micro operations” or “μops”
Does Multiple Issue Work?

• Yes, but not as much as we’d like
• Programs have real dependencies that limit ILP
• Some dependencies are hard to eliminate
  – e.g. pointer aliasing
• Some parallelism is hard to expose
  – Limited window size during instruction issue
• Memory delays and limited bandwidth
  – Hard to keep pipelines full
• Speculation can help if done well
Get To Know Your Instructor
Agenda

• Higher Level ILP
• Administrivia
• Dynamic Scheduling
• Virtual Memory Introduction
Memory Hierarchy

Earlier: Caches

Next Up: Virtual Memory

Upper Level
Faster

Lower Level

Memory

Disk

Tape
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 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 code starts at 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 $\rightarrow$ 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 that 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")

64 MB

Physical Memory

Virtual Memory

\[ \infty \]

Heap

Static

0

0
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
0x0000 page 0 4 Ki
0x1000 page 1 4 Ki
... ...
0x7000 page 7 4 Ki

Virtual Address
0x0000 page 0 4 Ki
0x01000 page 1 4 Ki
0x02000 page 2 4 Ki
... ...
0x1F000 page 31 4 Ki

Physical Memory
Virtual Memory
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)

---

Physical Page # | Page Offset
---|---
Not necessarily the same size

Virtual Page # | Page Offset
---|---
Same Size
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)

  ![Diagram showing Virtual Page # and Page Offset]

• **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

Virtual Address: [VPN] [offset]

1) Index into PT using VPN

2) Check Valid and Access Rights bits

3) Combine PPN and offset

Page Table

<table>
<thead>
<tr>
<th>V</th>
<th>AR</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>XX</td>
<td></td>
</tr>
</tbody>
</table>

4) Use PA to access memory

Physical Address
Page Table Entry Format

• Contains either PPN or indication not in main memory
• Valid = Valid page table entry
  – 1 → virtual page is in physical memory
  – 0 → 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
Paging/Virtual Memory Multiple Processes

User A:
Virtual Memory

Virtual Memory

User B:
Virtual Memory

Physical Memory

64 MB

Page Table A

Page Table B

Static

Code

Stack

Static

Code
# 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: 4Ki-8KiB</td>
</tr>
<tr>
<td>Placement:</td>
<td>Fully Associative</td>
</tr>
<tr>
<td>Direct Mapped,</td>
<td>(almost always)</td>
</tr>
<tr>
<td>N-way Set Associative</td>
<td></td>
</tr>
<tr>
<td>Replacement:</td>
<td>LRU</td>
</tr>
<tr>
<td>LRU or Random</td>
<td>Write Back</td>
</tr>
<tr>
<td>Write Thru or Back</td>
<td></td>
</tr>
</tbody>
</table>
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