Review: Pipeline Performance

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls
  + Data Hazard Stalls + Control Stalls
  - Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches, jumps, exceptions)
Review: Types of Data Hazards

Consider executing a sequence of
\[ r_k \leftarrow (r_i) \text{ op } (r_j) \]
type of instructions

Data-dependence
\[ r_3 \leftarrow (r_1) \text{ op } (r_2) \quad \text{Read-after-Write} \]
\[ r_5 \leftarrow (r_3) \text{ op } (r_4) \quad \text{(RAW) hazard} \]

Anti-dependence
\[ r_3 \leftarrow (r_1) \text{ op } (r_2) \quad \text{Write-after-Read} \]
\[ r_1 \leftarrow (r_4) \text{ op } (r_5) \quad \text{(WAR) hazard} \]

Output-dependence
\[ r_3 \leftarrow (r_1) \text{ op } (r_2) \quad \text{Write-after-Write} \]
\[ r_3 \leftarrow (r_6) \text{ op } (r_7) \quad \text{(WAW) hazard} \]

Data Hazards: An Example

\begin{align*}
I_1 & : \text{DIVD} & & \text{dest} & & \text{src1} & & \text{src2} \\
I_2 & : \text{LD} & & f_6, & & f_6, & & f_4 \\
I_3 & : \text{MULTD} & & f_2, & & 45(r_3) \\
I_4 & : \text{DIVD} & & f_0, & & f_2, & & f_4 \\
I_5 & : \text{SUBD} & & f_8, & & f_6, & & f_2 \\
I_6 & : \text{ADDD} & & f_{10}, & & f_0, & & f_6
\end{align*}

RAW Hazards
WAR Hazards
WAW Hazards
Pipelining becomes complex when we want high performance in the presence of:
- Long latency or partially pipelined floating-point units
- Multiple function and memory units
- Memory systems with variable access time
- Precise exceptions

How to prevent increased writeback latency from slowing down single cycle integer operations?

Bypassing
**Complex Pipeline**

Can we solve write hazards without equalizing all pipeline depths and without bypassing?

---

**When is it Safe to Issue an Instruction?**

Suppose a data structure keeps track of all the instructions in all the functional units.

The following checks need to be made before the Issue stage can dispatch an instruction:

- Is the required function unit available?
- Is the input data available? ⇒ RAW?
- Is it safe to write the destination? ⇒ WAR? WAW?
- Is there a structural conflict at the WB stage?
Scoreboard for In-order Issues

Busy[FU#] : a bit-vector to indicate FU’s availability.
(FU = Int, Add, Mult, Div)
These bits are hardwired to FU's.

WP[reg#] : a bit-vector to record the registers for which
writes are pending.
These bits are set to true by the Issue stage and set to
false by the WB stage.

Issue checks the instruction (opcode dest src1 src2)
against the scoreboard (Busy & WP) to dispatch

FU available? Busy[FU#]
RAW? WP[src1] or WP[src2]
WAR? cannot arise
WAW? WP[dest]

In-Order Issue Limitations: an example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD</td>
<td>F2, 34(R2)</td>
</tr>
<tr>
<td>2</td>
<td>LD</td>
<td>F4, 45(R3)</td>
</tr>
<tr>
<td>3</td>
<td>MULTD</td>
<td>F6, F4, F2</td>
</tr>
<tr>
<td>4</td>
<td>SUBD</td>
<td>F8, F2, F2</td>
</tr>
<tr>
<td>5</td>
<td>DIVD</td>
<td>F4, F2, F8</td>
</tr>
<tr>
<td>6</td>
<td>ADDD</td>
<td>F10, F6, F4</td>
</tr>
</tbody>
</table>

In-order: 1 (2,1) . . . . 2 3 4 4 3 5 . . . 5 6 6
In-order restriction prevents instruction 4
from being dispatched

(underline indicates cycle when instruction writes back)
**Out-of-Order Issue**

- Issue stage buffer holds multiple instructions waiting to issue.
- Decode adds next instruction to buffer if there is space and the instruction does not cause a WAR or WAW hazard.
- Any instruction in buffer whose RAW hazards are satisfied can be issued (for now at most one dispatch per cycle). On a write back (WB), new instructions may get enabled.

---

**In-Order Issue Limitations: an example**

<table>
<thead>
<tr>
<th>1</th>
<th>LD</th>
<th>F2, 34(R2)</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>LD</td>
<td>F4, 45(R3)</td>
<td>long</td>
</tr>
<tr>
<td>3</td>
<td>MULTD</td>
<td>F6, F4, F2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>SUBD</td>
<td>F8, F2, F2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>DIVD</td>
<td>F4, F2, F8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>ADDD</td>
<td>F10, F6, F4</td>
<td>1</td>
</tr>
</tbody>
</table>

In-order: 1 (2,1) . . . . . . 2 3 4 4 3 5 . . . 5 6 6
Out-of-order: 1 (2,1) 4 4 . . . . 2 3 . . . 3 5 . . . 5 6 6

*Out-of-order execution did not allow any significant improvement!*

---

9/11/2007 11
9/11/2007 12
How many instructions can be in the pipeline?

Which features of an ISA limit the number of instructions in the pipeline?

- **Number of Registers**

Which features of a program limit the number of instructions in the pipeline?

- **Control transfers**

Out-of-order dispatch by itself does not provide any significant performance improvement!

Overcoming the Lack of Register Names

Floating Point pipelines often cannot be kept filled with small number of registers.

- IBM 360 had only 4 Floating Point Registers

*Can a microarchitecture use more registers than specified by the ISA without loss of ISA compatibility?*

Robert Tomasulo of IBM suggested an ingenious solution in 1967 based on on-the-fly register renaming
Little’s Law

*Throughput (T) = Number in Flight (N) / Latency (L)*

Example:
- 4 floating point registers
- 8 cycles per floating point operation

⇒ maximum of ½ issue per cycle!

---

Instruction-level Parallelism via Renaming

<table>
<thead>
<tr>
<th>1</th>
<th>LD</th>
<th>F2, 34(R2)</th>
<th>latency 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>LD</td>
<td>F4, 45(R3)</td>
<td>long</td>
</tr>
<tr>
<td>3</td>
<td>MULTD</td>
<td>F6, F4, F2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>SUBD</td>
<td>F8, F2, F2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>DIVD</td>
<td>F4′, F2, F8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>ADDD</td>
<td>F10, F6, F4′</td>
<td>1</td>
</tr>
</tbody>
</table>

In-order: 1 (2,1) . . . . . . 2 3 4 4 3 5 . . . 5 6 6
Out-of-order: 1 (2,1) 4 4 5 . . . 2 (3,5) 3 6 6

Any antidependence can be eliminated by renaming. (renaming ⇒ additional storage)

Can it be done in hardware? yes!
Register Renaming

- Decode does register renaming and adds instructions to the issue stage reorder buffer (ROB)
  ⇒ renaming makes WAR or WAW hazards impossible

- Any instruction in ROB whose RAW hazards have been satisfied can be dispatched.
  ⇒ Out-of-order or dataflow execution

Dataflow execution

Instruction slot is candidate for execution when:
- It holds a valid instruction ("use" bit is set)
- It has not already started execution ("exec" bit is clear)
- Both operands are available (p1 and p2 are set)
Renaming & Out-of-order Issue
An example

Renaming table

<table>
<thead>
<tr>
<th>p</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>r9</td>
</tr>
<tr>
<td>F2</td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td>t5</td>
</tr>
<tr>
<td>F5</td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td>t3</td>
</tr>
<tr>
<td>F7</td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td>t4</td>
</tr>
</tbody>
</table>

data / \( t_i \)

Reorder buffer

<table>
<thead>
<tr>
<th>Ins#</th>
<th>use</th>
<th>exec</th>
<th>op</th>
<th>p1</th>
<th>src1</th>
<th>p2</th>
<th>src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>LD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>MUL</td>
<td>v2</td>
<td>1</td>
<td>v1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>SUB</td>
<td>v1</td>
<td>1</td>
<td>v1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>DIV</td>
<td>v1</td>
<td>0</td>
<td>v4</td>
<td></td>
</tr>
</tbody>
</table>

- When are names in sources replaced by data?
  Whenever an FU produces data
- When can a name be reused?
  Whenever an instruction completes

Data-Driven Execution

Renaming table & reg file

Reorder buffer

Replacing the tag by its value is an expensive operation

- Instruction template (i.e., tag \( t \)) is allocated by the Decode stage, which also stores the tag in the reg file
- When an instruction completes, its tag is deallocated
Simplifying Allocation/Deallocation

Instruction buffer is managed circularly
- "exec" bit is set when instruction begins execution
- When an instruction completes, its "use" bit is marked free
- \( \text{ptr}_2 \) is incremented only if the "use" bit is marked free

IBM 360/91 Floating Point Unit

*R. M. Tomasulo, 1967*

Common bus ensures that data is made available immediately to all the instructions waiting for it
Effectiveness?

Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but did not show up in the subsequent models until mid-Nineties.

Why?

Reasons
1. Effective on a very small class of programs
2. Memory latency a much bigger problem
3. Exceptions not precise!

One more problem needed to be solved

Control transfers

CS252 Administrivia

• Prereq quiz will be handed back Thursday at end of class
• Bspace experience?
• First reading assignment split into two parts, second part due Thursday
• Next reading assignment distributed Thursday, due Tuesday
• Please see me if you’re currently waitlisted
In the News… AMD Quad-Core

- Released 9/10/2007
- First single die quad-core x86 processor
- Major marketing points include energy efficiency & virtualization features (not clock rate)

Precise Interrupts

*It must appear as if an interrupt is taken between two instructions* (say \(I_i\) and \(I_{i+1}\))

- the effect of all instructions up to and including \(I_i\) is totally complete
- no effect of any instruction after \(I_i\) has taken place

The interrupt handler either aborts the program or restarts it at \(I_{i+1}\).
Effect on Interrupts

*Out-of-order Completion*

\[ I_1 \text{ DIVD } f6, f6, f4 \]
\[ I_2 \text{ LD } f2, 45(r3) \]
\[ I_3 \text{ MULTD } f0, f2, f4 \]
\[ I_4 \text{ DIVD } f8, f6, f2 \]
\[ I_5 \text{ SUBD } f10, f0, f6 \]
\[ I_6 \text{ ADDD } f6, f8, f2 \]

*Out-of-order comp* 1 2 2 3 1 4 3 5 5 4 6 6

Consider interrupts

*Precise interrupts are difficult to implement at high speed - want to start execution of later instructions before exception checks finished on earlier instructions*

---

Exception Handling

*(In-Order Five-Stage Pipeline)*

- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage
Phases of Instruction Execution

- **Fetch**: Instruction bits retrieved from cache.
- **Decode**: Instructions placed in appropriate issue (aka “dispatch”) stage buffer.
- **Execute**: Instructions and operands sent to execution units. When execution completes, all results and exception flags are available.
- **Commit**: Instruction irrevocably updates architectural state (aka "graduation" or "completion").

In-Order Commit for Precise Exceptions

- Instructions fetched and decoded into instruction reorder buffer in-order.
- Execution is out-of-order (⇒ out-of-order completion).
- Commit (write-back to architectural state, i.e., regfile & memory) is in-order.

Temporary storage needed to hold results before commit (shadow registers and store buffers)
Extensions for Precise Exceptions

- add <pd, dest, data, cause> fields in the instruction template
- commit instructions to reg file and memory in program order ⇒ buffers can be maintained circularly
- on exception, clear reorder buffer by resetting $ptr_1 = ptr_2$
  (stores must wait for commit before updating memory)

Rollback and Renaming

Register file does not contain renaming tags any more. How does the decode stage find the tag of a source register? Search the “dest” field in the reorder buffer
Renaming Table

Renaming table is a cache to speed up register name look up. It needs to be cleared after each exception taken. When else are valid bits cleared? **Control transfers**

9/11/2007

Branch Penalty

How many instructions need to be killed on a misprediction?

Modern processors may have > 10 pipeline stages between next pc calculation and branch resolution!
Average Run-Length between Branches

Average dynamic instruction mix from SPEC92:

<table>
<thead>
<tr>
<th></th>
<th>SPECint92</th>
<th>SPECfp92</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>39 %</td>
<td>13 %</td>
</tr>
<tr>
<td>FPU Add</td>
<td>20 %</td>
<td></td>
</tr>
<tr>
<td>FPU Mult</td>
<td>13 %</td>
<td></td>
</tr>
<tr>
<td>load</td>
<td>26 %</td>
<td>23 %</td>
</tr>
<tr>
<td>store</td>
<td>9 %</td>
<td>9 %</td>
</tr>
<tr>
<td>branch</td>
<td>16 %</td>
<td>8 %</td>
</tr>
<tr>
<td>other</td>
<td>10 %</td>
<td>12 %</td>
</tr>
</tbody>
</table>

SPECint92:  
compress, eqntott, espresso, gcc, li

SPECfp92:  
doduc, ear, hydro2d, mdijdp2, su2cor

What is the average run length between branches?

next lecture: Branch prediction & Speculative execution

---

Paper Discussion: B5000 vs IBM 360

- IBM set foundations for ISAs since 1960s
  - 8-bit byte
  - Byte-addressable memory (as opposed to word-addressable memory)
  - 32-bit words
  - Two’s complement arithmetic (but not the first processor)
  - 32-bit (SP) / 64-bit (DP) Floating Point format and registers
  - Commercial use of microcoded CPUs
  - Binary compatibility / computer family

- B5000 very different model: HLL only, stack, Segmented VM

- IBM paper made case for ISAs good for microcoded processors \( \Rightarrow \) leading to CISC