Last time in Lecture 11

- Modern page-based virtual memory systems provide:
  - Translation, Protection, Virtual memory.
- Translation and protection information stored in page tables, held in main memory
- Translation and protection information cached in “translation lookaside buffer” (TLB) to provide single cycle translation+protection check in common case
- VM interacts with cache design
  - Physical cache tags require address translation before tag lookup, or use untranslated offset bits to index cache.
  - Virtual tags do not require translation before cache hit/miss determination, but need to be flushed or extended with ASID to cope with context swaps. Also, must deal with virtual address aliases (usually by disallowing copies in cache).
Complex Pipelining: Motivation

Pipelining becomes complex when we want high performance in the presence of:

- Long latency or partially pipelined floating-point units
- Memory systems with variable access time
- Multiple arithmetic and memory units

Floating-Point Unit (FPU)

Much more hardware than an integer unit

Single-cycle FPU is a bad idea - why?

- it is common to have several FPU’s
- it is common to have different types of FPU’s \( Fadd, Fmul, Fdiv, \ldots \)
- an FPU may be pipelined, partially pipelined or not pipelined

To operate several FPU’s concurrently the FP register file needs to have more read and write ports
Functional Unit Characteristics

- **fully pipelined**
  - 1 cyc 1 cyc 1 cyc

- **partially pipelined**
  - 2 cyc 2 cyc

Functional units have internal pipeline registers

- Operands are latched when an instruction enters a functional unit.
- Inputs to a functional unit (e.g., register file) can change during a long latency operation.

Floating-Point ISA

Interaction between the floating-point datapath and the integer datapath is determined largely by the ISA.

**MIPS ISA**

- Separate register files for FP and Integer instructions.
  
  *The only interaction is via a set of move instructions (some ISA’s don’t even permit this)*

- Separate load/store for FPR’s and GPR’s but both use GPR’s for address calculation.

- Separate conditions for branches.
  
  FP branches are defined in terms of condition codes.
Realistic Memory Systems

Latency of access to the main memory is usually much greater than one cycle and often unpredictable. *Solving this problem is a central issue in computer architecture*

Common approaches to improving memory performance:
- separate instruction and data memory ports
  ⇒ *self-modifying code might need explicit cache flush*
- caches
  *single cycle except in case of a miss ⇒ stall*
- interleaved memory
  *multiple memory accesses ⇒ bank conflicts*
- split-phase memory operations
  ⇒ *out-of-order responses*

Multiple Functional Units in Pipeline

IF → ID → Issue → ALU → Mem → WB

GPR’s FPR’s

Fadd
Fmul
Fdiv
Complex Pipeline Control Issues

- Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
- Structural conflicts at the write-back stage due to variable latencies of different functional units
- Out-of-order write hazards due to variable latencies of different functional units
- How to handle exceptions?

Complex In-Order Pipeline

Delay writeback so all operations have same latency to W stage
- Write ports never oversubscribed (one inst. in & one inst. out every cycle)
- Stall pipeline on long latency operations, e.g., divides, cache misses
- Handle exceptions in-order at commit point

How to prevent increased writeback latency from slowing down single cycle integer operations? Bypassing
Types of Data Hazards

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

type of instructions

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

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

Output-dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \]
\[ r_3 \leftarrow r_6 \text{ op } r_7 \]
Write-after-Write (WAW) hazard
Register vs. Memory Dependence

Data hazards due to register operands can be determined at the decode stage but

data hazards due to memory operands can be determined only after computing the effective address

\[
\begin{align*}
\text{store} & \quad M[r_1 + \text{disp1}] \leftarrow r_2 \\
\text{load} & \quad r_3 \leftarrow M[r_4 + \text{disp2}]
\end{align*}
\]

Does \((r_1 + \text{disp1}) = (r_4 + \text{disp2})\)?

Data Hazards: An Example

RAW Hazards
WAR Hazards
WAW Hazards
Instruction Scheduling

Valid orderings:

in-order: $I_1, I_2, I_3, I_4, I_5, I_6$

out-of-order: $I_2, I_1, I_3, I_4, I_5, I_6$

out-of-order: $I_1, I_2, I_3, I_5, I_4, I_6$

Out-of-order Completion

In-order Issue

<table>
<thead>
<tr>
<th>$I_1$</th>
<th>DIVD</th>
<th>f6, f6, f4</th>
<th>Latency 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_2$</td>
<td>LD</td>
<td>f2, 45(r3)</td>
<td>1</td>
</tr>
<tr>
<td>$I_3$</td>
<td>MULTD</td>
<td>f0, f2, f4</td>
<td>3</td>
</tr>
<tr>
<td>$I_4$</td>
<td>DIVD</td>
<td>f8, f6, f2</td>
<td>4</td>
</tr>
<tr>
<td>$I_5$</td>
<td>SUBD</td>
<td>f10, f0, f6</td>
<td>1</td>
</tr>
<tr>
<td>$I_6$</td>
<td>ADDD</td>
<td>f6, f8, f2</td>
<td>1</td>
</tr>
</tbody>
</table>

in-order comp: 1 2 1 2 3 4 3 5 4 6 5 6
out-of-order comp: 1 2 3 1 4 3 5 4 6 6
IBM Memo on CDC6600

Thomas Watson Jr., IBM CEO, August 1963:

“Last week, Control Data ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers... Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer.”

To which Cray replied: “It seems like Mr. Watson has answered his own question.”
CS152 Administrivia

- Thursday Mar 19, Quiz 3
  - Virtual memory hierarchy lectures L9 - L11, PS 3, Lab 3

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?

A Data Structure for Correct Issues

*Keeps track of the status of Functional Units*

<table>
<thead>
<tr>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Dest</th>
<th>Src1</th>
<th>Src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Div</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The instruction *i* at the Issue stage consults this table:

- FU available? check the busy column
- RAW? search the dest column for *i*’s sources
- WAR? search the source columns for *i*’s destination
- WAW? search the dest column for *i*’s destination

An entry is added to the table if no hazard is detected; An entry is removed from the table after Write-Back.
Simplifying the Data Structure
Assuming In-order Issue

Suppose the instruction is not dispatched by the Issue stage if a RAW hazard exists or the required FU is busy, and that operands are latched by functional unit on issue:

Can the dispatched instruction cause a WAR hazard?
- **NO**: Operands read at issue
- **YES**: Out-of-order completion

No WAR hazard!  no need to keep src1 and src2
The Issue stage does not dispatch an instruction in case of a WAW hazard
- ⇒ a register name can occur at most once in the dest column

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
- ⇒ Each pipeline stage in the FU's must carry the dest field and a flag to indicate if it is valid “the (we, ws) pair”
### 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

|---------------|-----------|------|----------------------|------|----------|

### Scoreboard Dynamics

<table>
<thead>
<tr>
<th>Functional Unit Status</th>
<th>Registers Reserved for Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int(1), Add(1), Mult(3), Div(4)</td>
<td>WB</td>
</tr>
<tr>
<td>t0</td>
<td>I₁</td>
</tr>
<tr>
<td>t1</td>
<td>I₂ f2</td>
</tr>
<tr>
<td>t2</td>
<td>f0</td>
</tr>
<tr>
<td>t3</td>
<td>I₃ f0</td>
</tr>
<tr>
<td>t4</td>
<td>f0</td>
</tr>
<tr>
<td>t5</td>
<td>I₄ f0 f8</td>
</tr>
<tr>
<td>t6</td>
<td>f8</td>
</tr>
<tr>
<td>t7</td>
<td>I₅ f10</td>
</tr>
<tr>
<td>t8</td>
<td>f8 f10</td>
</tr>
<tr>
<td>t9</td>
<td>f8</td>
</tr>
<tr>
<td>t10</td>
<td>I₆ f6</td>
</tr>
<tr>
<td>t11</td>
<td></td>
</tr>
</tbody>
</table>

- **I₁** : DIVD, f6, f6, f4
- **I₂** : LD, f2, 45(r3)
- **I₃** : MULTD, f0, f2, f4
- **I₄** : DIVD, f8, f6, f2
- **I₅** : SUBD, f10, f0, f6
- **I₆** : ADDD, f6, f8, f2
In-Order Issue Limitations: an example

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

In-order restriction prevents instruction 4 from being dispatched

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.
  - Note: WAR possible again because issue is out-of-order (WAR not possible with in-order issue and latching of input operands at functional unit)
- 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.
**Issue Limitations: In-Order and Out-of-Order**

<table>
<thead>
<tr>
<th>Issue Limitations: In-Order and Out-of-Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. LD F2, 34(R2)</td>
</tr>
<tr>
<td>2. LD F4, 45(R3)</td>
</tr>
<tr>
<td>3. MULTD F6, F4, F2</td>
</tr>
<tr>
<td>4. SUBD F8, F2, F2</td>
</tr>
<tr>
<td>5. DIVD F4, F2, F8</td>
</tr>
<tr>
<td>6. ADDD F10, F6, F4</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!*

---

**How many instructions can be in the pipeline?**

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

*Number of Registers*

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 using on-the-fly *register renaming*

---

Instruction-level Parallelism via Renaming

<table>
<thead>
<tr>
<th>No</th>
<th>Opcode</th>
<th>1st Oper</th>
<th>2nd Oper</th>
<th>LATENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD</td>
<td>F2</td>
<td>34(R2)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>LD</td>
<td>F4</td>
<td>45(R3)</td>
<td>long</td>
</tr>
<tr>
<td>3</td>
<td>MULTD</td>
<td>F6, F4</td>
<td>F2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>SUBD</td>
<td>F8, F2</td>
<td>F2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>DIVD</td>
<td>F4’, F2</td>
<td>F8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>ADDD</td>
<td>F10, F6</td>
<td>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!
Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)

• MIT material derived from course 6.823
• UCB material derived from course CS252