Pipelining is complicated by multiple and/or variable latency functional units

Out-of-order and/or pipelined execution requires tracking of dependencies

- RAW
- WAR
- WAW

Dynamic issue logic can support out-of-order execution to improve performance
- Last time, looked at simple scoreboard to track out-of-order completion
In-Order Issue Pipeline

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

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.
### Issue Limitations: In-Order and Out-of-Order

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

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

\[ \text{Throughput (T)} = \frac{\text{Number in Flight (N)}}{\text{Latency (L)}} \]

Example:

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

\[ \Rightarrow \frac{1}{2} \text{ issues per cycle!} \]
Instruction-level Parallelism via Renaming

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD F2, 34(R2)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>LD F4, 45(R3)</td>
<td></td>
<td>long</td>
</tr>
<tr>
<td>3</td>
<td>MULTD F6, F4, F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>SUBD F8, F2, F2</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>DIVD F4’, F2, F8</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>ADDD F10, F6, F4’</td>
<td></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>v1</td>
</tr>
<tr>
<td>F2</td>
<td>v1</td>
</tr>
<tr>
<td>F3</td>
<td>v5</td>
</tr>
<tr>
<td>F4</td>
<td>v3</td>
</tr>
<tr>
<td>F5</td>
<td>v4</td>
</tr>
<tr>
<td>F6</td>
<td>v3</td>
</tr>
<tr>
<td>F7</td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td>v4</td>
</tr>
</tbody>
</table>

#### Reorder buffer

<table>
<thead>
<tr>
<th>Ins#</th>
<th>use 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>0</td>
<td>LD</td>
<td>v2</td>
<td>v1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>LD</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>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>SUB</td>
<td>v1</td>
<td>1</td>
<td>v1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>DIV</td>
<td>v1</td>
<td>0</td>
<td>v4</td>
</tr>
</tbody>
</table>

1. When are names in sources replaced by data?

   *Whenever an FU produces data*

2. When can a name be reused?

   *Whenever an instruction completes*
CS152 Administrivia

- Quiz 2 to be handed back at end of class
- Lab 3 due today (?)
- Quiz 3 on Tuesday
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
- ptr₂ is incremented only if the “use” bit is marked free
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
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

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

\( \text{out-of-order comp} \) 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

- **PC**
  - Fetch: Instruction bits retrieved from cache.

- **I-cache**
  - Decode: Instructions placed in appropriate issue (aka “dispatch”) stage buffer

- **Fetch Buffer**
  - Issue Buffer
    - Execute: Instructions and operands sent to execution units. When execution completes, all results and exception flags are available.

- **Func. Units**
  - Result Buffer
    - 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 ptr₁=ptr₂
  (stores must wait for commit before updating memory)
Rollback and Renaming

Register File (now holds only committed state)

Reorder buffer

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

Rename Table

Register File

Reorder buffer

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
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