Last time...

- Complex pipelining
  - Multiple functional units with variable access time
- Types of data hazards
  - RAW, WAR, WAW
- Dependency graph
  - How instructions are dependent on each other
  - Basis for out-of-order

---

Complex Pipeline

![Complex Pipeline Diagram](image)

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?

---

Types of Data Hazards

Consider executing a sequence of

\[ r_x = r_y \text{ op } r_z \]

type of instructions

- Data-dependence
  \[ r_x = r_y \text{ op } r_z \quad r_y = r_z \text{ op } r_w \]
  Read-after-Write (RAW) hazard

- Anti-dependence
  \[ r_x = r_y \text{ op } r_z \quad r_y = r_z \text{ op } r_w \]
  Write-after-Read (WAR) hazard

- Output-dependence
  \[ r_x = r_y \text{ op } r_z \quad r_y = r_z \text{ op } r_w \]
  Write-after-Write (WAW) hazard

---

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>biv</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  
WAW hazard?  
YES: Out-of-order completion

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]  
WAR? WP[src1] or WP[src2]  
WAW? cannot arise  
WAW? WP[dest]

CSE 490/590 Administrivia

• Midterm on Friday, 3/4  
• Review on Wednesday, 3/2  
• Project 1 deadline: Friday, 3/11  
• Office hours this week: Wed after class until 2pm

In-Order Issue Limitations: an example

In-order: 1 2 3 4 5 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

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

1 LD F2, 34(R2) latency 1
2 LD F4, 45(R3) long
3 MULTD F6, F4, F2 3
4 SUBD F8, F2, F2 1
5 DIVD F4, F2, F8 4
6 ADDO F10, F6, F4 1

In-order: 1 (2, 1) . . . . . . 2 3 4 4 5 . . . . 5 6 6
Out-of-order: 1 (2, 1) 4 4 . . . . . . 2 3 . . . . 5 . . . . 5 6 6
Any antidependence can be eliminated by renaming.
(renamming → additional storage)
Can it be done in hardware? yes!

Acknowledgements

• These slides heavily contain material developed and copyright by
  – Krste Asanovic (MIT/UCB)
  – David Patterson (UCB)
• And also by:
  – Arvind (MIT)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
• MIT material derived from course 6.823
• UCB material derived from course CS252