Complex Pipelining:
Out-of-Order Execution, Register Renaming and Exceptions

Joel Emer
Computer Science and Artificial Intelligence Laboratory
M.I.T.

http://www.csg.csail.mit.edu/6.823
CDC 6600-style Scoreboard

Instructions are issued in order; An instruction is issued only if

- It cannot cause a RAW hazard
  \[\text{if operands are read immediately then no need to remember sources of instructions in the execute phases}\]

- It cannot cause a WAW hazard
  \[\text{There can be at most one instruction in the execute phase that can write in a particular register}\]

Scoreboard:
Two bit-vectors

\begin{itemize}
  \item {Busy[FU#]: Indicates FU's availability}
    \begin{itemize}
      \item These bits are hardwired to FU's.
    \end{itemize}
  \item {WP[reg#]: Records if a write is pending for a register}
    \begin{itemize}
      \item Set to true by the Issue stage and set to false by the WB stage
    \end{itemize}
\end{itemize}
In-Order Issue Limitations: *an example*

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD</td>
<td>F2, 34(R2)</td>
<td>1</td>
</tr>
<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

In-order restriction prevents instruction 4 from being dispatched
Out-of-Order Issue

How can we address the delay caused by a RAW dependence associated with the next in-order instruction?

- 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.
- Can issue any instruction in buffer whose RAW hazards are satisfied (*for now at most one dispatch per cycle*). 
  
*Note:* A writeback (WB) may enable more instructions.

Find something else to do!
In-Order Issue Limitations: an example

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Registers</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD</td>
<td>F2, R2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>LD</td>
<td>F4, 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!
How many Instructions can be in the pipeline

Throughput limited by number of instructions in flight, but which feature of an ISA limits the number of instructions in the pipeline?

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

How can we better understand the impact of number of registers on throughput?
Little’s Law

Throughput ($T$) = Number in Flight ($\bar{N}$) / Latency ($\bar{L}$)

Example:

4 floating point registers
8 cycles per floating point operation

⇒ ½ issues per cycle!
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?

Yes, Robert Tomasulo of IBM suggested an ingenious solution in 1967 based on on-the-fly register renaming.
Instruction-level Parallelism via Renaming

Latency

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

Renaming eliminates WAR and WAW hazards
(replacing renaming with additional storage)
Handling register dependencies

- Decode does register renaming, providing a new spot for each register write

  $\Rightarrow$ Renaming eliminates structural hazards (WAR and WAW) by allowing use of more storage space.

- Renamed instructions added to an issue stage structure, called the reorder buffer (ROB). Any instruction in ROB whose RAW hazards have been satisfied can be dispatched.

  $\Rightarrow$ Out-of-order or dataflow execution handles RAW hazards
Reorder Buffer

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)

Is it obvious where an architectural register value is?  No
## Renaming & Out-of-order Issue

### Renaming table

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

### 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Holds data (v_i) or tag(t_i)

- When are names in sources replaced by data?
  Whenever an FU produces data

- When can a name be reused?
  Whenever an instruction completes
## 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></td>
</tr>
<tr>
<td>F2</td>
<td>v1</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>v4</td>
</tr>
</tbody>
</table>

#### 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>0</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>t4</td>
<td></td>
</tr>
</tbody>
</table>

- data (v_i) / tag(t_i)

- **Insert instruction in ROB**
- **Issue instruction from ROB**
- **Complete instruction**
- **Empty ROB entry**

```
1  LD  F2,  34(R2)
2  LD  F4,  45(R3)
3  MULTD F6, F4, F2
4  SUBD F8, F2, F2
5  DIVD F4, F2, F8
6  ADDD F10, F6, F4
```
Simplifying Allocation/Deallocation

Instruction buffer is managed circularly
- Set “exec” bit when instruction begins execution
- When an instruction completes its “use” bit is marked free
- Increment $\text{ptr}_2$ only if the “use” bit is marked free
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
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

store buffers (to memory)

distribute instruction templates by functional units

load buffers (from memory)

Floating Point Reg

Adder

Mult

< t, result >
Effectiveness?

Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but was effective only on a very small class of problems and thus did not show up in the subsequent models until mid-nineties.

Why?

1. Did not address the memory latency problem which turned out be a much bigger issue than FU latency
2. Made exceptions imprecise

One more problem needed to be solved

Control transfers

More on this in the next lecture
Precise Exceptions

Exceptions are relatively unlikely events that need special processing, but where adding explicit control flow instructions is not desired, e.g., divide by 0, page fault.

Exceptions can be viewed as an implicit conditional subroutine call that is inserted between two instructions.

Therefore, it must appear as if the exception is taken between two instructions (say $I_i$ and $I_{i+1}$).

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

The handler either aborts the program or restarts it at $I_{i+1}$. 
Effect on Exceptions

Out-of-order Completion

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

out-of-order comp 1 2 2 3 1 4 3 5 5 4 6 6

Consider exceptions on “DIVD”s

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

- Exceptions create a dependence on the value of the next PC

- Options for handling this dependence:
  - Stall: No
  - Bypass: No
  - Find something else to do: No
  - Change the architecture: Sometimes: Alpha, Multiflow
  - Speculate!: Most common approach!

- How can we handle rollback on mis-speculation

  Delay state update until commit on speculated instructions

- Note: earlier exceptions must override later ones
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”).
Exception Handling
(In-Order Five-Stage Pipeline)

Hold exception flags in pipeline until commit point (M stage)

• If exception at commit:
  • update Cause/EPC registers
  • kill all stages
  • fetch at handler PC

Inject external interrupts at commit point
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 \(<pd, \text{dest}, \text{data}, \text{cause}\>\) 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 \(\text{ptr}_1 = \text{ptr}_2\)
  
  \((\text{stores must wait for \text{commit} before \text{updating memory}})\)
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 is a cache to speed up register name lookup. It needs to be cleared after each exception taken. When else are valid bits cleared? Control transfers.
Physical Register Files

- Reorder buffers are space inefficient – a data value may be stored in multiple places in the reorder buffer
- Idea – keep all data values in a physical register file
  - Tag represents the name of the data value and name of the physical register that holds it
  - Reorder buffer contains only tags

Thus, 64 data values may be replaced by 8-bit tags for a 256 element physical register file

More on this in later lectures ...
Branch Penalty

How many instructions need to be killed on a misprediction?

Modern processors may have > 10 pipeline stages between nextPC calculation and branch resolution!

next lecture: Branch prediction & Speculative execution

March 8, 2017