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

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Instructions are issued in order. An instruction is issued only if
- It cannot cause a RAW hazard
- It cannot cause a WAW hazard

⇒ There can be at most instruction in the execute phase that can write in a particular register

WAR hazards are not possible
- Due to in-order issue + operands read immediately

Scoreboard:
Two bit-vectors

Busy[FU#]: Indicates FU’s availability
These bits are hardwired to FU's.

WP[reg#]: Records if a write is pending for a register
Set to true by the Issue stage and set to false by the WB stage
### Scoreboard Dynamics

#### Functional Unit Status

<table>
<thead>
<tr>
<th>Issue time</th>
<th>Int(1)</th>
<th>Add(1)</th>
<th>Mult(3)</th>
<th>Div(4)</th>
<th>WB</th>
<th>WP</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td></td>
<td></td>
<td>f6</td>
<td></td>
<td>f6</td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td></td>
<td>I2</td>
<td>f2</td>
<td>f6</td>
<td>f6, f2</td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td></td>
<td></td>
<td>f6</td>
<td>f2</td>
<td>f6, f2</td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>I3</td>
<td></td>
<td>f0</td>
<td>f6</td>
<td>f6, f0</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td></td>
<td></td>
<td>f0</td>
<td>f6</td>
<td>f6, f0</td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td>I4</td>
<td></td>
<td>f0, f8</td>
<td></td>
<td>f0, f8</td>
<td></td>
</tr>
<tr>
<td>t6</td>
<td></td>
<td></td>
<td>f8</td>
<td>f0</td>
<td>f0, f8</td>
<td></td>
</tr>
<tr>
<td>t7</td>
<td>I5</td>
<td></td>
<td>f10</td>
<td>f8</td>
<td>f8, f10</td>
<td></td>
</tr>
<tr>
<td>t8</td>
<td></td>
<td></td>
<td>f8</td>
<td>f10</td>
<td>f8, f10</td>
<td></td>
</tr>
<tr>
<td>t9</td>
<td></td>
<td></td>
<td>f8</td>
<td>f8</td>
<td>f8</td>
<td></td>
</tr>
<tr>
<td>t10</td>
<td>I6</td>
<td></td>
<td>f6</td>
<td></td>
<td>f6</td>
<td></td>
</tr>
<tr>
<td>t11</td>
<td></td>
<td></td>
<td>f6</td>
<td>f6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Issue time

- $I_1$: DIVD, f6, f6, f4
- $I_2$: LD, f2, 45(r3)
- $I_3$: MULTD, f0, f2, f4
- $I_4$: DIVD, f8, f6, f2
- $I_5$: SUBD, f10, f0, f6
- $I_6$: ADDD, f6, f8, f2
## In-Order Issue Limitations

**An example**

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

WAR hazard prevents instruction 5 from being dispatched.

Out-of-order execution did not produce a significant improvement!
How many Instructions can be in the pipeline

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

*Number of Registers*

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

How can we better understand the impact of number of registers on throughput?
Throughput ($\bar{T}$) = Number in Flight ($\bar{N}$) / Latency ($\bar{L}$)

Example:

4 floating point registers
8 cycles per floating point operation

⇒ $\frac{1}{2}$ 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

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

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

- Decode does register renaming, providing a new spot for each register write
  - Renaming eliminates WAR and WAW hazards by allowing use of more storage space

- Renamed instructions added to an issue stage structure, called the reorder buffer (ROB). Any instruction in the ROB whose RAW hazards have been satisfied can be dispatched
  - 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 ("present" bits p1 and p2 are set)

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

### Renaming table & reg file

<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>

- Holds data ($v_i$) or tag ($t_i$)

### 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>
</table>

- $t_1$
- $t_2$
- $t_3$
- $t_4$
- $t_5$

- **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 & reg file

<table>
<thead>
<tr>
<th>p</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>0</td>
</tr>
<tr>
<td>F3</td>
<td>0</td>
</tr>
<tr>
<td>F4</td>
<td>0</td>
</tr>
<tr>
<td>F5</td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td>0</td>
</tr>
<tr>
<td>F7</td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td>0</td>
</tr>
</tbody>
</table>

- data ($v_i$) / tag($t_i$)

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>LD</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>LD</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>MUL</td>
<td>0</td>
<td>$v_2$</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>SUB</td>
<td>1</td>
<td>$v_1$</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>DIV</td>
<td>1</td>
<td>$v_1$</td>
<td>0</td>
<td>$t_4$</td>
</tr>
</tbody>
</table>

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

March 5, 2020
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 ptr₂ only if the “use” bit is marked free
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.
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

Branch/jump penalties

More on this in the next lecture
Reminder: 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 Iᵢ and Iᵢ₊₁)

• the effect of all instructions up to and including Iᵢ is complete
• no effect of any instruction after Iᵢ has taken place

The handler either aborts the program or restarts it at Iᵢ₊₁.
## Effect on Exceptions

### Out-of-order Completion

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$</td>
<td>DIVD</td>
<td>$f_6, f_6, f_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_2$</td>
<td>LD</td>
<td>$f_2, 45(r_3)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_3$</td>
<td>MULTD</td>
<td>$f_0, f_2, f_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_4$</td>
<td>DIVD</td>
<td>$f_8, f_6, f_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_5$</td>
<td>SUBD</td>
<td>$f_{10}, f_0, f_6$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_6$</td>
<td>ADDD</td>
<td>$f_6, f_8, f_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**out-of-order comp**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**restore f2**

**restore f10**
Exceptions

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

• Options for handling this dependence:
  • Stall
  • Bypass
  • Find something else to do
  • Change the architecture
  • Speculate!

    No
    No
    No
    Sometimes: Alpha, Multiflow
    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
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
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**

1. PC
2. I-cache
3. Fetch Buffer
4. Issue Buffer
5. Func. Units
6. Results Buffer
7. Arch. State

**Out of order**

1. PC
2. I-cache
3. Fetch Buffer
4. Issue Buffer
5. Func. Units
6. Results Buffer
7. Arch. State
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 $\Rightarrow$ buffers can be maintained circularly.
- On exception, clear reorder buffer by resetting $\text{ptr}_1 = \text{ptr}_2$.

(Stores must wait for commit before updating memory.)
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 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-bit 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