Lecture 10 – Complex Pipelines, Out-of-Order Issue, Register Renaming

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152
Last time in Lecture 9

- 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

- Virtual memory 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).
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.

Store: \[ M[r_1 + \text{disp1}] \leftarrow r_2 \]
Load: \[ r_3 \leftarrow M[r_4 + \text{disp2}] \]

Does \((r_1 + \text{disp1}) = (r_4 + \text{disp2})\)?
Data Hazards: An Example

$I_1$  FDIV.D
$I_2$  FLD
$I_3$  FMUL.D
$I_4$  FDIV.D
$I_5$  FSUB.D
$I_6$  FADD.D

RAW Hazards
WAR Hazards
WAW Hazards
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
Issues in Complex Pipeline Control

- 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?
Recap: 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*
Can we solve write hazards without equalizing all pipeline depths and without bypassing?
# Instruction Scheduling

## Valid orderings:

### in-order

\[ I_1 \quad I_2 \quad I_3 \quad I_4 \quad I_5 \quad I_6 \]

### out-of-order

\[ I_2 \quad I_1 \quad I_3 \quad I_4 \quad I_5 \quad I_6 \]

\[ I_1 \quad I_2 \quad I_3 \quad I_5 \quad I_4 \quad I_6 \]
# Out-of-order Completion

*In-order Issue*

<table>
<thead>
<tr>
<th>$l_1$</th>
<th>FDIV.D</th>
<th>f6, f6, f4</th>
<th>Latency</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_2$</td>
<td>FLD</td>
<td>f2, 45(x3)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$l_3$</td>
<td>FMULT.D</td>
<td>f0, f2, f4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$l_4$</td>
<td>FDIV.D</td>
<td>f8, f6, f2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>$l_5$</td>
<td>FSUB.D</td>
<td>f10, f0, f6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$l_6$</td>
<td>FADD.D</td>
<td>f6, f8, f2</td>
<td>1</td>
<td></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 2 3 1 4 3 5 5 4 6 6
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*

WAW hazard?

*YES: Out-of-order completion*
Simplifying the Data Structure ...

- 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 by the Issue stage and cleared by the WB stage
  - Each pipeline stage in the FU's must carry the register destination field and a flag to indicate if it is valid
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 by Issue stage and cleared by 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]
## Scoreboard Dynamics

<table>
<thead>
<tr>
<th>t0</th>
<th>Functional Unit Status</th>
<th>Registers Reserved for Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>$I_1$</td>
<td>f6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td>$I_2$ f2</td>
<td>f6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td></td>
<td>f6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f6, f2</td>
</tr>
<tr>
<td>t3</td>
<td>$I_3$ f0</td>
<td>f6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f6, f0</td>
</tr>
<tr>
<td>t4</td>
<td></td>
<td>f6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f6, f0</td>
</tr>
<tr>
<td>t5</td>
<td>$I_4$ f0 f8</td>
<td>f0 f8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t6</td>
<td></td>
<td>f0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f0, f8</td>
</tr>
<tr>
<td>t7</td>
<td>$I_5$ f10</td>
<td>f8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f8, f10</td>
</tr>
<tr>
<td>t8</td>
<td></td>
<td>f8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f8, f10</td>
</tr>
<tr>
<td>t9</td>
<td></td>
<td>f8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f8</td>
</tr>
<tr>
<td>t10</td>
<td>$I_6$ f6</td>
<td>f6</td>
</tr>
</tbody>
</table>

### Instructions

- $I_1$: FDIV.D
- $I_2$: FLD
- $I_3$: FMULT.D
- $I_4$: FDIV.D
- $I_5$: FSUB.D
- $I_6$: FADD.D
In-Order Issue Limitations: *an example*

1. FLD  f2,  34(x2)  latency 1
2. FLD  f4,  45(x3)  long
3. FMULT.D  f6,  f4,  f2  3
4. FSUB.D  f8,  f2,  f2  1
5. FDIV.D  f4,  f2,  f8  4
6. FADD.D  f10,  f6,  f4  1

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

In-order issue restriction prevents instruction 4 from being dispatched
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>Step</th>
<th>Instruction</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FLD</td>
<td>f2, 34(x2)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>FLD</td>
<td>f4, 45(x3)</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>FMULT.Df6,</td>
<td>f4, f2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>FSUB.D</td>
<td>f8, f2, f2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>FDIV.D</td>
<td>f4, f2, f8</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>FADD.D</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 . . . . 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?

Out-of-order dispatch by itself does not provide any significant performance improvement!
CS152 Administrivia

- Midterm in class Monday March 2
  - Covers lectures 1 – 9, plus assigned problem sets, labs, book readings
  - Excludes this lecture

- Lab 2 due Monday March 9
CS252 Administrivia

- No discussion on Monday March 2 – midterm!
- Give a <5-minute presentation in class in discussion section time on March 9
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*
Issue Limitations: In-Order and Out-of-Order

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Register(s)</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FLD</td>
<td>f2,</td>
<td>34(x2)</td>
</tr>
<tr>
<td>2</td>
<td>FLD</td>
<td>f4,</td>
<td>45(x3)</td>
</tr>
<tr>
<td>3</td>
<td>FMULT.D</td>
<td>f6, f4, f2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>FSUB.D</td>
<td>f8, f2, f2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>FDIV.D</td>
<td>f4', f2, f8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>FADD.D</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 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 instruction 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
Renaming Structures

**Renaming**
- **table & regfile**

**Reorder buffer**

- Instruction template (i.e., tag t) is allocated by the Decode stage, which also associates tag with register in regfile
- When an instruction completes, its tag is deallocated

Replacing the tag by its value is an expensive operation
Reorder Buffer Management

### Instruction Slot Candidate for Execution

- 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)

### ROB Management

- "exec" bit is set when instruction begins execution
- When an instruction completes its "use" bit is marked free
- \( \text{ptr}_2 \) is incremented only if the "use" bit is marked free

### Table

<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>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( t_2 )</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>

### Diagram

- Destination registers are renamed to the instruction's slot tag
- \( \text{ptr}_1 \) next available
- \( \text{ptr}_2 \) next to deallocate
Renaming & Out-of-order Issue

An example

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

- When can a name be reused?
  Whenever an instruction completes

1. FLD  f2,       34(x2)
2. FLD  f4,       45(x3)
3. FMULT.D f6, f4,   f2
4. FSUB.D f8, f2,   f2
5. FDIV.D f4, f2,   f8
6. FADD.D f10, f6,   f4

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>t5</td>
</tr>
<tr>
<td>f4</td>
<td></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>t2</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>
Common bus ensures that data is made available immediately to all the instructions waiting for it. Match tag, if equal, copy value & set presence “p”.
IBM ACS

- Second supercomputer project (Y) started at IBM in response to CDC6600
- Multiple Dynamic instruction Scheduling (DIS) invented by Lynn Conway for ACS
  - Used unary encoding of register specifiers and wired-OR logic to detect any hazards (similar design used in Alpha 21264 in 1995!)
- Seven-issue, out-of-order processor
  - Two decoupled streams, each with DIS
- Cancelled in favor of IBM360-compatible machines
Out-of-Order Fades into Background

Out-of-order processing implemented commercially in 1960s, but disappeared again until 1990s as two major problems had to be solved:

- **Precise traps**
  - Imprecise *traps* complicate debugging and OS code
  - Note, precise *interrupts* are relatively easy to provide

- **Branch prediction**
  - Amount of exploitable instruction-level parallelism (ILP) limited by control hazards

Also, simpler machine designs in new technology beat complicated machines in old technology

- Big advantage to fit processor & caches on one chip
- Microprocessors had era of 1%/week performance scaling
In-Order Commit for Precise Traps

- In-order instruction fetch and decode, and dispatch to reservation stations inside reorder buffer
- Instructions issue from reservation stations out-of-order
- Out-of-order completion, values stored in temporary buffers
- Commit is in-order, checks for traps, and if none updates architectural state
Separating Completion from Commit

- Re-order buffer holds register results from completion until commit
  - Entries allocated in program order during decode
  - Buffers completed values and exception state until in-order commit point
  - Completed values can be used by dependents before committed (bypassing)
  - Each entry holds program counter, instruction type, destination register specifier and value if any, and exception status (info often compressed to save hardware)

- Memory reordering needs special data structures
  - Speculative store address and data buffers
  - Speculative load address and data buffers
Phases of Instruction Execution

- **Fetch**: Instruction bits retrieved from instruction cache.
- **Decode**: Instructions dispatched to appropriate issue buffer.
- **Execute**: Instructions and operands issued to functional units. When execution completes, all results and exception flags are available.
- **Commit**: Instruction irrevocably updates architectural state (aka “graduation”), or takes precise trap/interrupt.
In-Order versus Out-of-Order Phases

- Instruction fetch/decode/rename always in-order
  - Need to parse ISA sequentially to get correct semantics
  - Proposals for speculative OoO instruction fetch, e.g., Multiscalar. Predict control flow and data dependencies across sequential program segments fetched/decoded/executed in parallel, fixup if prediction wrong

- Dispatch (place instruction into machine buffers to wait for issue) also always in-order
  - Some use “Dispatch” to mean “Issue”, but not in these lectures
In-Order Versus Out-of-Order Issue

▪ In-order issue:
  – Issue stalls on RAW dependencies or structural hazards, or possibly WAR/WAW hazards
  – Instruction cannot issue to execution units unless all preceding instructions have issued to execution units

▪ Out-of-order issue:
  – Instructions dispatched in program order to reservation stations (or other forms of instruction buffer) to wait for operands to arrive, or other hazards to clear
  – While earlier instructions wait in issue buffers, following instructions can be dispatched and issued out-of-order
In-Order versus Out-of-Order Completion

- All but the simplest machines have out-of-order completion, due to different latencies of functional units and desire to bypass values as soon as available.

- Classic RISC 5-stage integer pipeline just barely has in-order completion:
  - Load takes two cycles, but following one-cycle integer op completes at same time, not earlier.
  - Adding pipelined FPU immediately brings OoO completion.
Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)