CS 152 Computer Architecture and Engineering

Lecture 13 - Out-of-Order Issue, Register Renaming, & Branch Prediction

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 12

- 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
- Hardware register renaming can further improve performance by removing hazards.
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.
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

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)
  \[\Rightarrow\] renaming makes WAR or WAW hazards impossible

- Any instruction in ROB whose RAW hazards have been satisfied can be dispatched.
  \[\Rightarrow\] 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

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

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

### Renaming table

<table>
<thead>
<tr>
<th>F1</th>
<th>p</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>v1</td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td>v1</td>
</tr>
<tr>
<td>F3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td></td>
<td>t5</td>
</tr>
<tr>
<td>F5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td></td>
<td>t3</td>
</tr>
<tr>
<td>F7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td></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>0</td>
<td>v2</td>
<td>1</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>SUB</td>
<td>1</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>1</td>
<td>v1</td>
<td>0</td>
<td>v4</td>
</tr>
</tbody>
</table>

Whenever an FU produces data

Whenever an instruction completes
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 associates tag with register in regfile
- 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
- ptr\(_2\) is incremented only if the “use” bit is marked free
IBM 360/91 Floating-Point Unit

R. M. Tomasulo, 1967

Distribute instruction templates by functional units

load buffers (from memory)

store buffers (to memory)

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”.

Instructions

Floating-Point Reg

Adder

Mult

< tag, result >
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Register 1</th>
<th>Register 2</th>
<th>Register 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$</td>
<td>DIVD</td>
<td>f6, f6, f4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_2$</td>
<td>LD</td>
<td>f2, 45(r3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_3$</td>
<td>MULTD</td>
<td>f0, f2, f4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_4$</td>
<td>DIVD</td>
<td>f8, f6, f2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_5$</td>
<td>SUBD</td>
<td>f10, f0, f6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I_6$</td>
<td>ADDD</td>
<td>f6, f8, f2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Consider interrupts
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

- **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 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 `ptr_1 = ptr_2` (stores must wait for commit before updating memory).

---

**Reorder buffer**

<table>
<thead>
<tr>
<th>Inst#</th>
<th>use</th>
<th>exec</th>
<th>op</th>
<th>p1</th>
<th>src1</th>
<th>p2</th>
<th>src2</th>
<th>pd</th>
<th>dest</th>
<th>data</th>
<th>cause</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`ptr_2` next to commit

`ptr_1` next available
Rollback and Renaming

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
Control Flow Penalty

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

How much work is lost if pipeline doesn’t follow correct instruction flow?

~ Loop length x pipeline width

March 4, 2010
CS152, Spring 2010
**MIPS Branches and Jumps**

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JR</td>
<td>After Inst. Decode</td>
<td>After Reg. Fetch</td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td>After Reg. Fetch*</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>

*Assuming zero detect on register read
## Branch Penalties in Modern Pipelines

UltraSPARC-III instruction fetch pipeline stages (in-order issue, 4-way superscalar, 750MHz, 2000)

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>PC Generation/Mux</td>
</tr>
<tr>
<td>P</td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>F</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>B</td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td>I</td>
<td>Complete Decode</td>
</tr>
<tr>
<td>J</td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td>R</td>
<td>Register File Read</td>
</tr>
<tr>
<td>E</td>
<td>Integer Execute</td>
</tr>
</tbody>
</table>

**Branch**  
**Target Address Known**

**Branch Direction & Jump**  
**Register Target Known**

Remainder of execute pipeline (+ another 6 stages)
Reducing Control Flow Penalty

Software solutions

- *Eliminate branches - loop unrolling*
  Increases the run length
- *Reduce resolution time - instruction scheduling*
  Compute the branch condition as early as possible (of limited value)

Hardware solutions

- Find something else to do - *delay slots*
  Replaces pipeline bubbles with useful work (requires software cooperation)
- *Speculate - branch prediction*
  *Speculative execution* of instructions beyond the branch
Branch Prediction

Motivation:
Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

Required hardware support:
Prediction structures:
• Branch history tables, branch target buffers, etc.

Mispredict recovery mechanisms:
• Keep result computation separate from commit
• Kill instructions following branch in pipeline
• Restore state to state following branch
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
   bne0 (preferred taken)  beq0 (not taken)

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64
   typically reported as ~80% accurate
Dynamic Branch Prediction
"learning based on past behavior"

Temporal correlation
The way a branch resolves may be a good predictor of the way it will resolve at the next execution.

Spatial correlation
Several branches may resolve in a highly correlated manner (a preferred path of execution)
Branch Prediction Bits

- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!

BP state:
\[(\text{predict take/¬take}) \times (\text{last prediction right/wrong})\]
Branch History Table

4K-entry BHT, 2 bits/entry, ~80-90% correct predictions
Exploiting Spatial Correlation

Yeh and Patt, 1992

\[
\text{if } (x[i] < 7) \text{ then } \\
\quad y += 1; \\
\text{if } (x[i] < 5) \text{ then } \\
\quad c -= 4;
\]

If first condition false, second condition also false

\textit{History register}, H, records the direction of the last N branches executed by the processor
Two-Level Branch Predictor

*Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)*

Fetch PC

2-bit global branch history shift register

Shift in Taken/¬Taken results of each branch

Taken/¬Taken?
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