Computer Architecture

Out-of-order Execution

By Yoav Etsion
With acknowledgement to Dan Tsafrir, Avi Mendelson, Lihu Rappoport, and Adi Yoaz
The need for speed: Superscalar

- Remember our goal: minimize CPU Time
  \[
  \text{CPU Time} = \text{duration of clock cycle} \times \text{CPI} \times \text{IC}
  \]

- So far we have learned that in order to
  - Minimize \textit{clock cycle} \implies \text{add more pipe stages}
  - Minimize \textit{CPI} \implies \text{utilize pipeline}
  - Minimize \textit{IC} \implies \text{change/improve the architecture}

- Why not make the pipeline deeper and deeper?
  - Beyond some point, adding more pipe stages doesn’t help, because
    - Control/data hazards increase, and become costlier
      - (Recall that in a pipelined CPU, CPI=1 only w/o hazards)

- So what can we do next?
  - Reduce the CPI by utilizing ILP (instruction level parallelism)
  - We will need to duplicate HW for this purpose...
A simple superscalar CPU

- Duplicates the pipeline to accommodate ILP (IPC > 1)
  - ILP = instruction-level parallelism

- Note that duplicating HW in just one pipe stage doesn’t help
  - e.g., when having 2 ALUs,
  - the bottleneck moves to other stages

- Conclusion:
  - Getting IPC > 1 requires to fetch/decode/exe/retire > 1 instruction per clock:
Example: Pentium® Processor

- Pentium fetches & decodes 2 instructions per cycle

- Before register file read, decide on *pairing*
  - Can the two instructions be executed in parallel? (yes/no)

- Pairing decision is based...
  - On data dependencies (instructions must be independent)
  - On resources (v-pipe can only execute some of the instructions; and also, some instruction use resources from both pipes)
Mis prediction penalty for superscalar CPUs

• MPI = miss-per-instruction ; MPR = miss-predict rate

\[
MPI = \frac{\text{#incorrectly predicted branches}}{\text{total # of instructions}} = \frac{\text{MPR}}{\text{total # of instructions}} \times \frac{\text{predicted branches}}{\text{total # of instructions}}
\]

• MPI correlates well with performance; e.g., assume:
  - MPR = 5%, branches/instructions = 20% ⇒ MPI = 1%
  - Without hazards, IPC=2 (2 instructions per cycles)
  - Flush penalty of 10 cycles

• We get:
  - MPI = 1% ⇒ flush in every 100 instructions, but
  - IPC=2 ⇒ flush every 100/2 = 50 cycles
  - 10 cycles flush penalty every 50 cycles
  - 20% in performance

• (For IPC=1, we would have had
  - 10 cycles flush penalty per 100 cycles ⇒ 10% in performance)
Is superscalar good enough?

• A superscalar processor can fetch, decode, execute, and retire, e.g., 2 instructions in parallel

• But…
  - Can execute only independent instructions in parallel
    - Whereas adjacent instructions are often dependent
  - So the utilization of the second pipe is often low

• Solution: out-of-order execution
  - Execute instructions based on the “data flow” graph, (rather than program order)
  - Still need to keep the semantics of the original program
Out-of-order in a nutshell

- **HW examines a sliding window of consecutive instructions**
  - The “instruction window”

- **Ready instructions get picked up from window**

- **Executed out of program order**

- **Instruction results are committed to the machine state (memory+reg. file) in original program order**
  - Why?

- **User is unaware (except that the program runs faster)**
Superscalar basics: Data flow analysis

• Example:

(1) \( r_1 \leftarrow r_4 / r_7 \) /* assume division takes 20 cycles */
(2) \( r_8 \leftarrow r_1 + r_2 \)
(3) \( r_5 \leftarrow r_5 + 1 \)
(4) \( r_6 \leftarrow r_6 - r_3 \)
(5) \( r_4 \leftarrow r_5 + r_6 \)
(6) \( r_7 \leftarrow r_8 * r_4 \)

In-order execution

| 1 | 2 | 3 | 4 | 5 | 6 |

In-order (2-way superscalar)

| 1 | 2 | 4 | 5 | 6 |
|---|
| 3 |

Out-of-order execution

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
OoO – general scheme

- **Fetch & decode in order**
  - Multiple instructions are fetched/decoded in parallel
  - Insts. put in reservation stations (RS)

- **Execute instructions that are ready in the reservation stations**
  - Instruction operands must be ready
  - Available execution resources

- **Following execution:**
  - Broadcast result on bypass network
  - Signal all dependent instructions that data is ready

- **Commit instructions in-order**
  - Can commit an instruction only after all preceding instructions (in program order) have committed
Out of order execution (OoO)

- **Advantages: Better performance!**
  - Exploit Instruction Level Parallelism (ILP)
  - Hide latencies (e.g., L1 data cache miss, divide)

- **Disadvantages:**
  - HW is much more complex than that of in-order processors

- **Can compilers do this work?**
  - In a very limited way – can only *statically* schedule instructions (VLIW)
  - Compilers lack *runtime* information
    - Conditional branch direction (compiler limited to basic blocks)
    - Data values, which may affect calculation time and control
    - Cache miss / hit

The key is dynamic analysis and resolution of data dependencies
OoO: data dependencies

- Example:

  (1) \( r_8 \leftarrow 20 \)
  (2) \( r_1 \leftarrow \text{addr}_1 \)
  (3) \( r_2 \leftarrow \text{addr}_2 \)
  (4) \( r_3 \leftarrow \text{addr}_3 \)

**LOOP:**

  (5) \( r_4 \leftarrow \text{MEM}[r_1] \)
  (6) \( r_1 \leftarrow r_1 + 4 \)
  (7) \( r_5 \leftarrow \text{MEM}[r_2] \)
  (8) \( r_2 \leftarrow r_2 + 4 \)
  (9) \( r_6 \leftarrow r_4 + r_5 \)
  (10) \( \text{MEM}[r_3] \leftarrow r_6 \)
  (11) \( r_3 \leftarrow r_3 + 4 \)
  (12) \( r_8 \leftarrow r_8 - 1 \)
  (13) \( \text{bnz } r_8, \text{LOOP} \)

Instruction dependence graph

Are all dependencies equal?
Data dependency types (I)

- **True dependence: **\textit{RaW} (Read-after-Write)
  
  (7) \[ r5 \leftarrow \text{MEM}[r2] \]
  
  (9) \[ r6 \leftarrow r4 + r5 \]

  - An instruction consumes data that was produced by an earlier instruction

- Can we eliminate such dependencies?
  
  - Not without a time machine... (or value speculation)
RaW examples

(1) \( r_8 \leftarrow 20 \)
(2) \( r_1 \leftarrow \text{addr1} \)
(3) \( r_2 \leftarrow \text{addr2} \)
(4) \( r_3 \leftarrow \text{addr3} \)

**LOOP:**
(5) \( r_4 \leftarrow \text{MEM}[r_1] \)
(6) \( r_1 \leftarrow r_1 + 4 \)
(7) \( r_5 \leftarrow \text{MEM}[r_2] \)
(8) \( r_2 \leftarrow r_2 + 4 \)
(9) \( r_6 \leftarrow r_4 + r_5 \)
(10) \( \text{MEM}[r_3] \leftarrow r_6 \)
(11) \( r_3 \leftarrow r_3 + 4 \)
(12) \( r_8 \leftarrow r_8 - 1 \)
(13) \( \text{bnz } r_8, \text{LOOP} \)
Data dependency types (II)

- **Anti-dependence:** *WaR* (Write-after-Read)
  
  \[(5) \quad r4 \leftarrow \text{MEM}[r1]\]
  
  \[(6) \quad r1 \leftarrow r1 + 4\]

- **False dependence:** *WaW* (Write-after-Write)

  \[(7) \quad r5 \leftarrow \text{MEM}[r2]\]
  
  \[(7*) \quad r5 \leftarrow \text{MEM}[r2] \quad // \quad * \quad \text{next iteration}\]

- **Can we eliminate such dependencies?**
  
  - Yes! if we divert the second write to an alternate storage location
  
  - Also known as *Register Renaming*
WaR examples

(1) \( r8 \leftarrow 20 \)
(2) \( r1 \leftarrow \text{addr1} \)
(3) \( r2 \leftarrow \text{addr2} \)
(4) \( r3 \leftarrow \text{addr3} \)

LOOP:
(5) \( r4 \leftarrow \text{MEM}[r1] \)
(6) \( r1 \leftarrow r1 + 4 \)
(7) \( r5 \leftarrow \text{MEM}[r2] \)
(8) \( r2 \leftarrow r2 + 4 \)
(9) \( r6 \leftarrow r4 + r5 \)
(10) \( \text{MEM}[r3] \leftarrow r6 \)
(11) \( r3 \leftarrow r3 + 4 \)
(12) \( r8 \leftarrow r8 - 1 \)
(13) \text{bnz} ~ r8, \text{LOOP}
**WaW examples**

1st iteration:
(5) \( r4 \leftarrow \text{MEM}[r1] \)
(6) \( r1 \leftarrow r1 + 4 \)
(7) \( r5 \leftarrow \text{MEM}[r2] \)
(8) \( r2 \leftarrow r2 + 4 \)
(9) \( r6 \leftarrow r4 + r5 \)
(10) \( \text{MEM}[r3] \leftarrow r6 \)
(11) \( r3 \leftarrow r3 + 4 \)
(12) \( r8 \leftarrow r8 - 1 \)
(13) \( \text{bnz } r8, \text{LOOP} \)

2nd iteration:
(5) \( r4 \leftarrow \text{MEM}[r1] \)
(6) \( r1 \leftarrow r1 + 4 \)
(7) \( r5 \leftarrow \text{MEM}[r2] \)
(8) \( r2 \leftarrow r2 + 4 \)
(9) \( r6 \leftarrow r4 + r5 \)
(10) \( \text{MEM}[r3] \leftarrow r6 \)
(11) \( r3 \leftarrow r3 + 4 \)
(12) \( r8 \leftarrow r8 - 1 \)
(13) \( \text{bnz } r8, \text{LOOP} \)
OoO: Main ingredients

• **Wide fetch/issue/decode/commit**
  - If only one inst. goes into the pipeline each cycle, then on average only one inst. will commit each cycle \( \Rightarrow \) IPC=1

• **Branch prediction**
  - Not much parallelism in basic blocks (insts. seq. between branches)
  - Identify ILP across branch (and loop) boundaries

• **Register renaming**
  - Break False- and Anti-dependencies

• **Speculative execution**
  - Speculate branch outcome without affecting correctness
OoO Pipeline

- **Fetch**
  - Branch prediction

- **Decode**
  - Register renaming

- **Reservation stations (RS)**
  - Instructions wait for the inputs
  - Instructions wait for functional units

- **Functional units (FU)**

- **Bypass network**
  - Broadcast computed values back to reservation stations and PRF

- **Reorder buffer (ROB)**
  - De-speculate execution, mostly by Committing instructions *in-order*

- **The instruction window is instantiated as RS & ROB**
Benefits of Register Renaming

1st iteration:
(5) \( r4 \leftarrow MEM[r1] \)
(6) \( r1 \leftarrow r1 + 4 \)
(7) \( r5 \leftarrow MEM[r2] \)
(8) \( r2 \leftarrow r2 + 4 \)
(9) \( r6 \leftarrow r4 + r5 \)
(10) \( MEM[r3] \leftarrow r6 \)
(11) \( r3 \leftarrow r3 + 4 \)
(12) \( r8 \leftarrow r8 - 1 \)
(13) \( bnz \ r8, \ LOOP \)

2nd iteration:
(5) \( r4 \leftarrow MEM[r1] \)
(6) \( r1 \leftarrow r1 + 4 \)
(7) \( r5 \leftarrow MEM[r2] \)
(8) \( r2 \leftarrow r2 + 4 \)
(9) \( r6 \leftarrow r4 + r5 \)
(10) \( MEM[r3] \leftarrow r6 \)
(11) \( r3 \leftarrow r3 + 4 \)
(12) \( r8 \leftarrow r8 - 1 \)
(13) \( bnz \ r8, \ LOOP \)

Critical path: 8 instructions

Remove WaW
Remove WaR
Result…
Benefits of Register Renaming

1st iteration:
(5) r4 ← MEM[r1]
(6) r1 ← r1 + 4
(7) r5 ← MEM[r2]
(8) r2 ← r2 + 4
(9) r6 ← r4 + r5
(10) MEM[r3] ← r6
(11) r3 ← r3 + 4
(12) r8 ← r8 - 1
(13) bnz r8, LOOP

2nd iteration:
(5) r4 ← MEM[r1]
(6) r1 ← r1 + 4
(7) r5 ← MEM[r2]
(8) r2 ← r2 + 4
(9) r6 ← r4 + r5
(10) MEM[r3] ← r6
(11) r3 ← r3 + 4
(12) r8 ← r8 - 1
(13) bnz r8, LOOP

New critical path: 4 instructions!
Register renaming: How does it work?

- Data is stored in a physical register file (PRF)

- **Architected register file (ARF) holds pointers to PRF registers**
  - Each register in ARF represents a register in the ISA
  - Registers in ARF point to the latest version of the datum in PRF
  - An instruction that writes to a register triggers a “rename” operation
    - Allocate new PRF register
    - Update pointer in ARF

- Naturally, PRF > ARF

- Note: Other methods to implement register renaming have been proposed in the past...
Register renaming: Example

Original code:

(5) \( r4 \leftarrow \text{MEM}[r1] \)
(6) \( r1 \leftarrow r1 + 4 \)
(7) \( r5 \leftarrow \text{MEM}[r2] \)
(8) \( r2 \leftarrow r2 + 4 \)
(9) \( r6 \leftarrow r4 + r5 \)
(10) \( \text{MEM}[r3] \leftarrow r6 \)
(11) \( r3 \leftarrow r3 + 4 \)
(12) \( r8 \leftarrow r8 - 1 \)
(13) \( \text{bnz } r8, \text{ LOOP} \)

Post-decode (as seen by RS):

(5) \( pr5 \leftarrow \text{MEM}[40] \)
(6) \( pr6 \leftarrow 40 + 4 \)
(7) \( pr7 \leftarrow \text{MEM}[120] \)
(8) \( pr8 \leftarrow 120 + 4 \)
(9) \( pr9 \leftarrow pr5 + pr7 \)
(10) \( \text{MEM}[200] \leftarrow pr9 \)
(11) \( pr10 \leftarrow 200 + 4 \)
(12) \( pr11 \leftarrow 20 - 1 \)
(13) \( \text{bnz } pr11, \text{ LOOP} \)
OoO Pipeline: Execution

- **Fetch**
  - Branch prediction

- **Decode**
  - Register renaming

- **Reservation stations (RS)**
  - Instructions wait for the inputs
  - Instructions wait for functional units

- **Functional units (FU)**

- **Bypass network**
  - Broadcast computed values back to reservation stations and PRF

- **Reorder buffer (ROB)**
  - De-speculate execution, mostly by Committing instructions **in-order**

- The instruction window is instantiated as RS & ROB
Out-of-order execution

- **Insts. registered in ROB**
  - ROB acts like a cyclic buffer

- **Decoded insts. sent to RS**
  - If operands are ready, inst. is sent to FU
  - Otherwise, listen on bypass network and wait for operands
  - Values sent on bypass network are tagged by physical Register

- **Executed insts. are marked in ROB as completed**
  - Computed value is sent over bypass network to consumers
OoO execution example

Instructions waiting in reservation stations:

- (5) \( pr5 \leftarrow \text{MEM}[40] \)
- (6) \( pr6 \leftarrow 40 + 4 \)
- (7) \( pr7 \leftarrow \text{MEM}[120] \)
- (8) \( pr8 \leftarrow 120 + 4 \)
- (9) \( pr9 \leftarrow pr5 + pr7 \)
- (10) \( \text{MEM}[200] \leftarrow pr9 \)
- (11) \( pr10 \leftarrow 200 + 4 \)
- (12) \( pr11 \leftarrow 20 - 1 \)
- (13) \( \text{bnz } pr11, \text{LOOP} \)

Instructions execute as soon as their operands become ready, rather than in program order.

- Broadcast \( pr5 \) & \( pr8 \)
- (9) receives \( pr5 \)
- Broadcast \( pr6 \) & \( pr7 \)
- (9) receives \( pr7 \)
- Broadcast \( pr9 \) & \( pr11 \)
- (10) receives \( pr9 \)
- (13) receives \( pr11 \)
OoO Pipeline: ROB & de-speculation

- **Fetch**
  - Branch prediction

- **Decode**
  - Register renaming

- **Reservation stations (RS)**
  - Instructions wait for the inputs
  - Instructions wait for functional units

- **Functional units (FU)**

- **Bypass network**
  - Broadcast computed values back to reservation stations and PRF

- **Reorder buffer (ROB)**
  - De-speculate execution, mostly by Committing instructions **in-order**

- **The instruction window is instantiated as RS & ROB**
Managing speculative execution

- Insts. must not affect machine state while they are speculative

- Mis-predicted paths need to be flushed

- Precise interrupts
  - Traps/Exceptions/Interrupts leave pipeline in well-known state
  - As if the offending instruction just executed

- Renamed registers must not be freed until a path is validated
  - In practice, ARF is saved (checkpoint) whenever the decoder encounters a branch instruction
Managing speculative execution

• **Common implementation:**
  - Fetch/Decode instructions from the predicted execution path
  - Instructions can **execute** as soon as their operands become ready
  - Instructions can **graduate and commit** to memory only once it is certain they should have been executed
    - An instruction commits only when all previous (in-order) instructions have committed ⇒ instructions commit in-order
    - Instructions on a mis-predicted execution path are flushed
Example: Managing speculation

- ROB contains both normal and speculative insts.
  - Some have already executed

- Can we commit any?

- Remember: some insts. might fail
  - Memory faults/exceptions
  - Divide-by-zero
  - Cannot commit younger insts., even if branches were resolved

- Only the oldest executed instructions can commit
  - Multiple insts. per cycle (n-way)

Instructions in ROB

1. \( pr5 \leftarrow MEM[40] \)
2. \( pr6 \leftarrow 40 + 4 \)
3. \( pr7 \leftarrow MEM[120] \)
4. \( pr8 \leftarrow 120 + 4 \)
5. \( pr9 \leftarrow pr5 + pr7 \)
6. \( MEM[200] \leftarrow pr9 \)
7. \( pr10 \leftarrow 200 + 4 \)
8. \( pr11 \leftarrow 20 - 1 \)
9. \( bnz \ pr11, LOOP \)
Scalability of Speculative Execution

• Examining a large instruction window requires highly accurate branch prediction

• Example:
  ❖ Window size: 150 insts.
  ❖ 30 branches to fill a window (avg. of branch every 5 instruction)
  
  ❖ **Case 1**: Prediction rate=95%
    • Probability to predict 30 branches: $0.95^{30} = 0.22$

  ❖ **Case 2**: Prediction rate=98%
    • Probability to predict 30 branches: $0.98^{30} = 0.55$

  ❖ **Case 2**: Prediction rate=99%
    • Probability to predict 30 branches: $0.99^{30} = 0.74$
OoO scalability: VLSI considerations

- Many large ported arrays
  - Register files (ARF and PRF)
  - Reservation stations
  - For example, a 4-way OoO pipeline requires:
    - Reg. files with 8 RD ports and 4 WR ports (decode width)
    - RS and ROB with 4 ports each (execute/commit width)

- More logic is needed, and it is more complex
  - Examples:
    - Register renaming
    - Wakeup logic in RS (which instructions are selected to run?)

- All reservation stations must be checked whenever a FU broadcasts a tagged result
  - Many, many comparators
OoO scalability: VLSI considerations

- **Very wide buses**
  - Multiple results sent on the bypass network on each cycle

- **Timing is a challenge - need additional pipe stages**
  - Rename analysis
  - Rename sources
  - Access available sources from committed register file
  - Allocate entry in reservation station
  - “Ready” Decision

**Balancing the machine is essential and complex**
OoO summary

• **Advantages**
  - Help exploit Instruction Level Parallelism (ILP)
  - Help hide latencies (e.g., cache miss, divide)
  - Superior/complementary to inst. Scheduler in the compiler
    - Dynamic instruction window

• **Complex micro-architecture**
  - Complex wakeup logic (instruction scheduler)
  - Requires reordering mechanism (retirement) in the back-end for:
    - Precise interrupt resolution
    - Misprediction/speculation recovery

• **Speculative Execution**
  - Advantage: larger scheduling window ⇒ reveals more ILP
  - Issues:
    - Complex logic needed to recover from mis-prediction
    - Runtime cost incurred when recovering from a mis-prediction
OoO summary

• First appeared in floating point unit of IBM mainframes
  ❖ Tomasulo’s algorithm, published in 1967

• Generalized by Patt, Hwu and Shebanow [1985]
  ❖ After that, quickly adopted by industry
    • DEC Alpha, Intel Pentium Pro

• Today it is ubiquitous:
  ❖ Intel: 4-way OoO; instruction windows up to 150-200 insts.
  ❖ AMD: 4-way OoO; instruction windows of ~70 insts.
  ❖ ARM (Cortex-A9/A15): 2/5-way OoO; instruction window 40-100+
    • Many ARM implementations exist...

• Numerous variations and optimizations and extensions have been studied, and are used in commercial products
OOO Processor Example

THE P6 MICROARCHITECTURE
The P6 family (i686)

- Features
  - 1\textsuperscript{st} out of order x86 (\Rightarrow data flow analysis)
  - Speculative execution (across branches; requires flush+recovery)
  - Multiple branch prediction (wide op window contains 5 branch on avg)
  - Register renaming (solves false dependencies, gives more regs)
  - Super-pipeline: \sim 12 pipe stages (P-IV had 31! i7 back to 14)

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Freq (MHz)</th>
<th>Bus (MHz)</th>
<th>L2 cache</th>
<th>Feature size**</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium® Pro</td>
<td>1995</td>
<td>150~200</td>
<td>60/66</td>
<td>256/512K*</td>
<td>0.5, 0.35\mu m</td>
</tr>
<tr>
<td>Pentium® II</td>
<td>1997</td>
<td>233~450</td>
<td>66/100</td>
<td>512K*</td>
<td>0.35, 0.25\mu m</td>
</tr>
<tr>
<td>Pentium® III</td>
<td>1999</td>
<td>450~1400</td>
<td>100/133</td>
<td>256/512K</td>
<td>0.25, 0.18, 0.13\mu m</td>
</tr>
<tr>
<td>Pentium® M</td>
<td>2003</td>
<td>900~2260</td>
<td>400/533</td>
<td>1M / 2M</td>
<td>0.13, 90nm</td>
</tr>
<tr>
<td>Core™</td>
<td>2005</td>
<td>1660~2330</td>
<td>533/667</td>
<td>2M</td>
<td>65nm</td>
</tr>
<tr>
<td>Core™ 2</td>
<td>2006</td>
<td>1800~2930</td>
<td>800/1066</td>
<td>2/4/8M</td>
<td>65nm</td>
</tr>
</tbody>
</table>

*off die

** size of smallest part is smaller than the feature size
The P6 family (i686)

- Was used until 2011:
  - MacBook Air (1.4GHz Core 2 Duo)
  - Due to relative low power consumption
- Clock frequency ~proportional to feature size
- After P-III came P-IV… which wasn’t ideal for mobile computing
- Much (not all) of the improvement comes from feature size minimization

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Freq (MHz)</th>
<th>Bus (MHz)</th>
<th>L2 cache</th>
<th>Feature size**</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium® Pro</td>
<td>1995</td>
<td>150~200</td>
<td>60/66</td>
<td>256/512K*</td>
<td>0.5, 0.35µm</td>
</tr>
<tr>
<td>Pentium® II</td>
<td>1997</td>
<td>233~450</td>
<td>66/100</td>
<td>512K*</td>
<td>0.35, 0.25µm</td>
</tr>
<tr>
<td>Pentium® III</td>
<td>1999</td>
<td>450~1400</td>
<td>100/133</td>
<td>256/512K</td>
<td>0.25, 0.18, 0.13µm</td>
</tr>
<tr>
<td>Pentium® M</td>
<td>2003</td>
<td>900~2260</td>
<td>400/533</td>
<td>1M / 2M</td>
<td>0.13, 90nm</td>
</tr>
<tr>
<td>Core™</td>
<td>2005</td>
<td>1660~2330</td>
<td>533/667</td>
<td>2M</td>
<td>65nm</td>
</tr>
<tr>
<td>Core™ 2</td>
<td>2006</td>
<td>1800~2930</td>
<td>800/1066</td>
<td>2/4/8M</td>
<td>65nm</td>
</tr>
</tbody>
</table>

*off die

** size of smallest part is smaller than the feature size
Chip logically partitioned to 3

• **Front end**
  – In order, get and ops from memory
  – Decode them + turn them
    – from CISC ops
    – to >=1 u-ops (RISC-like)
  – So x86 input=CISC, but internally it’s actually RISC
  – The front-end is responsible for making the transition

• **Core**
  – Out of order, speculative, superscalar, renames registers

• **Retire**
  – In order
  – Commits when speculation ends
  – Can simultaneously commit up to 3 ops ("width" of machine)
P6 µArch

• In-Order Front End
  – BIU: Bus Interface Unit
  – IFU: Instruction Fetch Unit (includes IC)
  – BPU: Branch Prediction Unit
  – ID: Instruction Decoder
  – MS: Micro-Instruction Sequencer
  – RAT: Register Alias Table

• Out-of-order Core
  – ROB: Reorder Buffer
  – RRF: Real Register File
  – RS: Reservation Stations
  – IEU: Integer Execution Unit
  – FEU: Floating-point Execution Unit
  – AGU: Address Generation Unit
  – MIU: Memory Interface Unit
  – DCU: Data Cache Unit
  – MOB: Memory Order Buffer
  – L2: Level 2 cache

• In-Order Retire

External Bus

BIU — MOB — ROB

IFU — DCU — MIU

BPU — AGU — IEU — FEU

MS — RAT

L2
**P6 µArch**

**In-Order Front End**

- **BIU**: Bus Interface Unit (fetches instructions)
- **IFU**: Instruction Fetch Unit (includes i-cache)
- **BPU**: Branch Prediction Unit
- **ID**: Instruction Decoder
- **MS**: Micro-Instruction Sequencer (complex ops are comprised of a sequence of µ-ops; simple ops are comprised of only 1 µ-op)
- **RAT**: Register Alias Table (solves false dep.; most recent arch => physical mapping)
Out-of-order Core

- L2: Level 2 cache
- MOB: Memory Order Buffer
- DCU: Data Cache Unit
- MIU: Memory Interface Unit
- AGU: Address Generation Unit
- RRF: “Real” Register File (not shown; the machine’s state)
- IEU: Integer Execution Unit
- FEU: Floating-point Execution Unit
- RS: Reservation Stations (All those ops whose dependencies aren’t yet met; up to 20; 5 ports to exe units)
- ROB: Reorder Buffer (The physical regs; one entry per op – the reg is the dest of the op; in order!)
P6 pipeline - 12 stages (10<=P6<=14)

1: Next IP
2: ICache lookup
3: ILD (instruction length decode)
4: rotate
5: ID1 (instruction decoded 1)
6: ID2 (instruction decoded 2)
7: RAT - rename sources &
   ALLOC - assign destinations
[O1] 8: ROB - read sources
      RS - schedule data-ready uops for dispatch
[O2] 9: RS - dispatch uops
[R1] 10: EX
[R2] 11-12: Retirement
### In-order front-end

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Instructions</th>
<th>uops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next IP Mux</td>
<td>BPU</td>
<td>MS</td>
</tr>
<tr>
<td>IFU</td>
<td>ILD</td>
<td>ID</td>
</tr>
<tr>
<td>IQ</td>
<td>IDQ</td>
<td></td>
</tr>
</tbody>
</table>

- **BPU** – Branch Prediction Unit – predict next fetch address
- **IFU** – Instruction Fetch Unit
  - iTLB translates virtual to physical address (next lecture)
  - ICache supplies 16byte/cyc (on miss: access L2, maybe memory)
- **ILD** – Instruction Length Decode – split bytes to instructions
- **IQ** – Instruction Queue – buffer the instructions
- **ID** – Instruction Decode – decode instructions into uops
- **MS** – Micro-Sequencer – provides uops for complex instructions
- **IDQ** – Instruction Decode Queue – buffer the uops
Branch prediction

• Implementation
  – Use local history to predict direction
  – Need to predict multiple branches
  ⇒ Need to predict branches before previous branches are resolved
  ⇒ Branch history updated first based on prediction, later based on actual execution (speculative history)
  – Target address taken from BTB

• Prediction rate: ~92%
  – High prediction rate is crucial for long pipelines
  – Especially important for OOOE, speculative execution:
    ➢ On misprediction all instructions following the branch in the instruction window are flushed
    ➢ Effective size of the window is determined by prediction accuracy
Branch prediction – clustering

• Given a fetched line (bytes), need to know which line to fetch next
  – Perhaps there’s more than one branch in the line
  – We must use 1\textsuperscript{st} (leftmost) taken branch (\geq the current fetched IP)

• Implemented by
  – Splitting IP into $\text{setOfLine} + \text{tagOfLine} + \text{offsetWithinLine}$
  – If there’s a match
    ➢ The offsets of the matching ways are ordered
    ➢ Ways with offset smaller than the fetch IP offset are discarded
    ➢ The 1st branch that’s predicted taken is chosen as the predicted branch
P6 BTB

- 2-level, local histories, per-set counters
- 4-way set associative: 512 entries in 128 sets

- Up to 4 branches can have the same set/tag match (since there are 4 ways)
In-order front-end – decoder

16 Instruction bytes from IFU

Instruction Length Decode

IQ

D0  D1  D2  D3
≤4 uops 1 uop 1 uop 1 uop

Determine where each x86 op starts

Buffers ops in queue

Convert ops into uops
- D0 handles complex ops (translate to up to 4 uops)
- D1-3 for simple ops that translate to 1 uop
- In case ops aligned with D1-3 decode into >1 uops, defer them to following cycle (whereby D0 will be used)

Buffers uops
- Smooth decoder’s variable throughput (we want the same number of uops every cycle (=4))
Micro operations (uops)

• Each CISC inst is broken into one or more RISC uops
  – Simplicity
    ➢ Each uop is (relatively) simple
    ➢ Canonical representation of src/dest (2 src, 1 dest)
  – But increased instruction count

• Simple instructions translate to a few uops
  – Typical uop count (not necessarily cycle count!)
    Reg-Reg ALU/Mov inst: 1 uop
    Mem-Reg Mov (load) 1 uop
    Mem-Reg ALU (load + op) 2 uops
    Reg-Mem Mov (store) 2 uops (st addr, st data)
    Reg-Mem ALU (ld + op + st) 4 uops

• Complex instructions translate into more uops
Out-of-order core: ROB + RS

- **Reorder Buffer (ROB):**
  - Holds all “not yet retired” instructions
  - 40 ordered entries (cyclic array)
  - Retired in-order
  - It’s possible some instruction already executed (their result known), but cannot be retired since
    - still have speculative status
    - and/or are waiting for previous instructions to retire in order

- **Reservation Stations (RS):**
  - Holds “not yet executed” instructions
  - 20 entries (subset of ROB)
  - Up to 4 simultaneous ops can get in and out of RS simultaneously

- **After execution**
  - Results written to both ROB & possibly to RS (when source of other instructions)
Out-of-order core: execution units

- **Basic execution units** (*EU; *=I|F)
  - IEU: Integer Execution Unit
  - FEU: Floating-point Execution Unit
- **Memory-related execution units**
  - DCU: Data Cache Unit
  - AGU: Address Generation Unit
    - Computes effective address & sends to DCU to fetch from cache)
  - MOB: Orders memory loads and stores
  - L2: Level 2 cache
  - MIU: Memory Interface Unit
- **MIU feeds execution units**
  - Positioned between RS & exe units
  - Has 5 ports (see next slide)
  - Some units may be similar between ports, others exclusive
Out-of-order core: execution units

2nd bypass in RS

1st bypass in MIU

internal 0-dealy bypass within each EU

Bypass: shorten path of source to exe unit, e.g., if current outcome of a certain exe unit is its source for next round, can use outcome directly, etc.
RAT & ALLOC

• There are $\leq 4$ new uops/cyc; for each such uop
  – Perform register allocation & renaming. Specifically…
• For each new uop, use RAT (Register Alias Table) to
  – Source reg(s): map arch reg(s) to physical reg(s)
    ➢ arch reg => latest phys reg that updated arch reg
  – Target reg: (1) allocate new phys reg; (2) update RAT accordingly
    ➢ Now arch reg points to newly allocated phys reg (for next time)

<table>
<thead>
<tr>
<th>arch reg</th>
<th>phys reg#</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0</td>
<td>RRF</td>
</tr>
<tr>
<td>EBX</td>
<td>19</td>
<td>ROB</td>
</tr>
<tr>
<td>ECX</td>
<td>23</td>
<td>ROB</td>
</tr>
</tbody>
</table>

• The Allocator (Alloc)
  – Assigns each uop with new ROB & RS entries
  – Write up the matching phys regs to RS (along with the rest of the uop)
  – Allocate Load & Store buffers in the MOB (for load & store ops)
Reorder buffer (ROB)

• Holds 40 uops which are “not yet committed”
  – Same order as program (cyclic array)
  – Provides large physical register space for reg renaming
  – A physical register is actually an item within a matching ROB entry
    ➢ phys reg number = ROB entry number
    ➢ phys reg = uop’s target destination (there’s always exactly one)
    ➢ phys regs buffer the execution results until retirement

<table>
<thead>
<tr>
<th>#entry</th>
<th>entryValid</th>
<th>dataValid</th>
<th>data (physical reg)</th>
<th>arch target reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>12H</td>
<td>EBX</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>33H</td>
<td>ECX</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>xxx</td>
<td>ESI</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td>xxx</td>
<td>XXX</td>
</tr>
<tr>
<td>39</td>
<td>0</td>
<td>0</td>
<td>xxx</td>
<td>XXX</td>
</tr>
</tbody>
</table>

– \textit{Valid data} is set after uop executed (& result written to physical reg)
**RRF – real register file**

- **Holds the Architectural Register File**
  - Architectural registers are numbered: 0 = EAX, 1 = EBX, …
  - This is “the state” of the chip (can’t roll back)

- **The value of an architectural register**
  - Is the value written to it by the last committed uop (which writes to that reg)
  - So long as we don’t change the RRF, we don’t change the state

<table>
<thead>
<tr>
<th>#entry</th>
<th>Arch Reg Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (EAX)</td>
<td>9AH</td>
</tr>
<tr>
<td>1 (EBX)</td>
<td>F34H</td>
</tr>
</tbody>
</table>

RRF:
Uop flow through the ROB

- Uops are entered in order (there’s a head and a tail)
  - Registers renamed by the entry #

- Once assigned
  - Execution order unimportant, only dependencies

- After execution:
  - Entries marked “executed” (*dataValid*=1) & wait for retirement
  - Retirement occurs once all prior instruction have retired
  - \(\Rightarrow\) Commit architectural state only after speculation was resolved

- Retirement
  - Detect exceptions and misprediction
    - Branch result might impact uops down the road
    - Initiate repair to get machine back on track
  - Update “real” regs (in RRF) with value of renamed (phys) regs
  - Update memory
  - Clear ROB entry
Reservation station (RS)

- Pool of all “not yet executed” uops
  - Holds the uop code & source data (until it is dispatched=scheduled)

- When a uop is allocated in RS, operand values are updated
  - If operand is arch reg => value taken from the RRF
  - If operand is phys reg (with dataValid =1) => value taken from ROB
  - If operand is phys reg (with dataValid=0) => wait for value

- The RS maintains operands status “ready / not-ready”
  - Each cycle, executed uops make more operands “ready”
    - RS arbitrates WB busses between exe units
    - RS monitors WB bus to capture data needed by waiting uops
    - Data can bypass directly from WB bus to exe unit (like we’ve seen)
  - Uops whose operands are ready (all of them) can be dispatched
    - Dispatcher chooses which ready uop to execute next
    - Dispatcher sends chosen uops to appropriate functional units
    - (Of course, need said appropriate functional units to be vacant)
Register Renaming example

Add EAX, EBX, EAX

RAT / Alloc

ROB37 = ROB19 + RRF0

RS

<table>
<thead>
<tr>
<th>#</th>
<th>Data Valid</th>
<th>Data</th>
<th>DST</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>V</td>
<td>V</td>
<td>12H</td>
</tr>
<tr>
<td>23</td>
<td>V</td>
<td>V</td>
<td>33H</td>
</tr>
<tr>
<td>37</td>
<td>I</td>
<td>x</td>
<td>xxx</td>
</tr>
<tr>
<td>38</td>
<td>I</td>
<td>x</td>
<td>xxx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Data Valid</th>
<th>Data</th>
<th>DST</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>V</td>
<td>V</td>
<td>12H</td>
</tr>
<tr>
<td>23</td>
<td>V</td>
<td>V</td>
<td>33H</td>
</tr>
<tr>
<td>37</td>
<td>V</td>
<td>I</td>
<td>xxx</td>
</tr>
<tr>
<td>38</td>
<td>I</td>
<td>x</td>
<td>xxx</td>
</tr>
</tbody>
</table>

RRF: 0 EAX 97H

IDQ
Register Renaming example

Add EAX, EBX, EAX

ROB37 = ROB19 + RRF0

allocate new ROB entry (phys reg) for EAX
Register Renaming example

Add EAX, EBX, EAX

ROB37 = ROB19 + RRF0

Update RAT (alias table) accordingly
### Register Renaming example

#### IDQ

#### RAT / Alloc

<table>
<thead>
<tr>
<th>#reg</th>
<th>RAT / Alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0 RRF</td>
</tr>
<tr>
<td>EBX</td>
<td>19 ROB</td>
</tr>
<tr>
<td>ECX</td>
<td>23 ROB</td>
</tr>
</tbody>
</table>

#### ROB

<table>
<thead>
<tr>
<th>#</th>
<th>Data Valid</th>
<th>Data</th>
<th>DST</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>V</td>
<td>V</td>
<td>12H EBX</td>
</tr>
<tr>
<td>23</td>
<td>V</td>
<td>V</td>
<td>33H ECX</td>
</tr>
<tr>
<td>37</td>
<td>I</td>
<td>x</td>
<td>xxx XXX</td>
</tr>
<tr>
<td>38</td>
<td>I</td>
<td>x</td>
<td>xxx XXX</td>
</tr>
</tbody>
</table>

#### RRF:

| 0 | EAX | 97H |

#### ROB37 = ROB19 + RRF0

#### RS

```
<table>
<thead>
<tr>
<th>#</th>
<th>Data Valid</th>
<th>Data</th>
<th>DST</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>V</td>
<td>V</td>
<td>12H EBX</td>
</tr>
<tr>
<td>23</td>
<td>V</td>
<td>V</td>
<td>33H ECX</td>
</tr>
<tr>
<td>37</td>
<td>V</td>
<td>I</td>
<td>xxx EAX</td>
</tr>
<tr>
<td>38</td>
<td>I</td>
<td>x</td>
<td>xxx XXX</td>
</tr>
</tbody>
</table>
```

#### Add EAX, EBX, EAX

Update RS accordingly
Register Renaming example (2)

sub EAX, ECX, EAX

RAT / Alloc

<table>
<thead>
<tr>
<th>#reg</th>
<th>EAX</th>
<th>37</th>
<th>ROB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EBX</td>
<td>19</td>
<td>ROB</td>
</tr>
<tr>
<td></td>
<td>ECX</td>
<td>23</td>
<td>ROB</td>
</tr>
</tbody>
</table>

ROB38 = ROB23 - ROB37

IDQ

ROB

<table>
<thead>
<tr>
<th>#</th>
<th>Data Valid</th>
<th>Data</th>
<th>DST</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>V</td>
<td>V</td>
<td>12H</td>
</tr>
<tr>
<td>23</td>
<td>V</td>
<td>V</td>
<td>33H</td>
</tr>
<tr>
<td>37</td>
<td>I</td>
<td>x</td>
<td>xxx</td>
</tr>
<tr>
<td>38</td>
<td>I</td>
<td>x</td>
<td>xxx</td>
</tr>
</tbody>
</table>

RRF: 0 EAX 97H

RS

<table>
<thead>
<tr>
<th></th>
<th>v</th>
<th>src1</th>
<th>v</th>
<th>src2</th>
<th>Pdst</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1</td>
<td>97H</td>
<td>1</td>
<td>12H</td>
<td>37</td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>rob37</td>
<td>1</td>
<td>33H</td>
<td>38</td>
</tr>
</tbody>
</table>
## Register Renaming example (2)

The example illustrates out-of-order execution by showing how register renaming is handled in a computer architecture context.

### RAT / Alloc

<table>
<thead>
<tr>
<th>#reg</th>
<th>EAX</th>
<th>37</th>
<th>ROB</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBX</td>
<td>19</td>
<td>ROB</td>
<td></td>
</tr>
<tr>
<td>ECX</td>
<td>23</td>
<td>ROB</td>
<td></td>
</tr>
</tbody>
</table>

After the `sub EAX, ECX, EAX` instruction is added to the IDQ, the following table shows the updated RAT/Alloc:

<table>
<thead>
<tr>
<th>#reg</th>
<th>EAX</th>
<th>38</th>
<th>ROB</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBX</td>
<td>19</td>
<td>ROB</td>
<td></td>
</tr>
<tr>
<td>ECX</td>
<td>23</td>
<td>ROB</td>
<td></td>
</tr>
</tbody>
</table>

### ROB

<table>
<thead>
<tr>
<th>#</th>
<th>Data Valid</th>
<th>Data</th>
<th>DST</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>V</td>
<td>V</td>
<td>12H</td>
</tr>
<tr>
<td>23</td>
<td>V</td>
<td>V</td>
<td>33H</td>
</tr>
<tr>
<td>37</td>
<td>I</td>
<td>x</td>
<td>xxx</td>
</tr>
<tr>
<td>38</td>
<td>I</td>
<td>x</td>
<td>xxx</td>
</tr>
</tbody>
</table>

After the instruction is executed, ROB entries are updated as follows:

<table>
<thead>
<tr>
<th>#</th>
<th>Data Valid</th>
<th>Data</th>
<th>DST</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>V</td>
<td>V</td>
<td>12H</td>
</tr>
<tr>
<td>23</td>
<td>V</td>
<td>V</td>
<td>33H</td>
</tr>
<tr>
<td>37</td>
<td>I</td>
<td>V</td>
<td>xxx</td>
</tr>
<tr>
<td>38</td>
<td>I</td>
<td>V</td>
<td>xxx</td>
</tr>
</tbody>
</table>

### RS

<table>
<thead>
<tr>
<th>v</th>
<th>src1</th>
<th>v</th>
<th>src2</th>
<th>Pdst</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1</td>
<td>97H</td>
<td>1</td>
<td>12H</td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>rob37</td>
<td>1</td>
<td>33H</td>
</tr>
</tbody>
</table>

### RRF

| 0 | EAX | 97H |

The instruction `sub EAX, ECX, EAX` is added to the IDQ. ROB38 is calculated as ROB23 - ROB37. Allocate another new ROB entry (phys reg) for EAX.
Register Renaming example (2)

**IDQ**

```
sub EAX, ECX, EAX
```

**RAT / Alloc**

<table>
<thead>
<tr>
<th>#reg</th>
<th>EAX 37</th>
<th>ROB</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBX 19</td>
<td>ROB</td>
<td></td>
</tr>
<tr>
<td>ECX 23</td>
<td>ROB</td>
<td></td>
</tr>
</tbody>
</table>

**ROB**

<table>
<thead>
<tr>
<th>#</th>
<th>Data Valid</th>
<th>Data</th>
<th>DST</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>V</td>
<td>V</td>
<td>12H</td>
</tr>
<tr>
<td>23</td>
<td>V</td>
<td>V</td>
<td>33H</td>
</tr>
<tr>
<td>37</td>
<td>I</td>
<td>x</td>
<td>xxx</td>
</tr>
<tr>
<td>38</td>
<td>I</td>
<td>x</td>
<td>xxx</td>
</tr>
</tbody>
</table>

**RS**

```
add 1      97H 1 12H 37
sub 0      rob37 1 33H 38
```

**Update RAT (alias table) accordingly**

ROB38 = ROB23 - ROB37
Register Renaming example (2)

\[
\begin{align*}
\text{sub EAX, ECX, EAX} & \\
\text{ROB38} & = \text{ROB23} - \text{ROB37}
\end{align*}
\]