Evolution of Intel Processor Pipelines
- 486, Pentium, Pentium Pro

Superscalar Processor Design
- Speculative Execution
- Register Renaming
- Branch Prediction

Intel x86 Processors

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
</tr>
<tr>
<td>486</td>
<td>1989</td>
<td>1.9M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
</tr>
<tr>
<td>Pentium/MMX</td>
<td>1997</td>
<td>4.5M</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
</tr>
<tr>
<td>Pentium M</td>
<td>2003</td>
<td>140M</td>
</tr>
<tr>
<td>Core Duo</td>
<td>2006</td>
<td>151M</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>291M</td>
</tr>
</tbody>
</table>

Architectural Performance

Metric
- SpecX92/MHz: Normalizes with respect to clock speed

Sampling

<table>
<thead>
<tr>
<th>Processor</th>
<th>MHz</th>
<th>SpecInt92</th>
<th>IntAP</th>
<th>SpecFP92</th>
<th>FltAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>i386/387</td>
<td>33</td>
<td>6</td>
<td>0.2</td>
<td>3</td>
<td>0.1</td>
</tr>
<tr>
<td>i486DX</td>
<td>50</td>
<td>28</td>
<td>0.6</td>
<td>13</td>
<td>0.3</td>
</tr>
<tr>
<td>Pentium</td>
<td>150</td>
<td>181</td>
<td>1.2</td>
<td>125</td>
<td>0.8</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>200</td>
<td>320</td>
<td>1.6</td>
<td>283</td>
<td>1.4</td>
</tr>
<tr>
<td>MIPS R3000A</td>
<td>25</td>
<td>16.1</td>
<td>0.6</td>
<td>21.7</td>
<td>0.9</td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>200</td>
<td>300</td>
<td>1.5</td>
<td>600</td>
<td>3.0</td>
</tr>
<tr>
<td>Alpha 21164a</td>
<td>417</td>
<td>500</td>
<td>1.2</td>
<td>750</td>
<td>1.8</td>
</tr>
</tbody>
</table>

i486 Pipeline

Fetch
- Load 16-bytes of instruction into prefetch buffer

Decode1
- Determine instruction length, instruction type

Decode2
- Compute memory address
- Generate immediate operands

Execute
- Register Read
- ALU operation
- Memory read/write

Write-Back
- Update register file
Pipeline Stage Details

Fetch
- Moves 16 bytes of instruction stream into code queue
- Not required every time
  - About 5 instructions fetched at once
  - Only useful if don’t branch
- Avoids need for separate instruction cache

D1
- Determine total instruction length
  - Signals code queue aligner where next instruction begins
- May require two cycles
  - When multiple operands must be decoded
  - About 6% of “typical” DOS program

Stage Details (Cont.)

D2
- Extract memory displacements and immediate operands
- Compute memory addresses
  - Add base register, and possibly scaled index register
- May require two cycles
  - If index register involved, or both address & immediate operand
  - Approx. 5% of executed instructions

EX
- Read register operands
- Compute ALU function
- Read or write memory (data cache)

WB
- Update register result

Data Hazards

Data Hazards
<table>
<thead>
<tr>
<th>Generated</th>
<th>Used</th>
<th>Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>ALU</td>
<td>EX–EX Forwarding</td>
</tr>
<tr>
<td>Load</td>
<td>ALU</td>
<td>EX–EX Forwarding</td>
</tr>
<tr>
<td>ALU</td>
<td>Store</td>
<td>EX–EX Forwarding</td>
</tr>
<tr>
<td>ALU</td>
<td>Eff. Address</td>
<td>(Stall) + EX-ID2 Forwarding</td>
</tr>
</tbody>
</table>

Control Hazards

Jump Instruction Processing
- Continue pipeline assuming branch not taken
- Resolve branch condition in EX stage
- Also speculatively fetch at target during EX stage
Control Hazards (Cont.)

Branch Not Taken
- Allow pipeline to continue.
- Total of 1 cycle for instruction.

Branch taken
- Flush instructions in pipe.
- Begin ID1 at target.
- Total of 3 cycles for instruction.

Comparison to 386

Cycles Per Instruction

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>386 Cycles</th>
<th>486 Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Store</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ALU</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Jump taken</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Jump not taken</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Call</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

Reasons for Improvement
- On chip cache
  - Faster loads & stores
- More pipelining

Comparison with Our pAlpha Pipeline

Two Decoding Stages
- Harder to decode CISC instructions
- Effective address calculation in D2

Multicycle Decoding Stages
- For more difficult decodings
- Stalls incoming instructions

Combined Mem/EX Stage
- Avoids load stall without load delay slot
  - But introduces stall for address computation
Pentium Pipeline

- Fetch & Align Instruction
- Decode Instruction
- Generate Control Word
- Decode Control Word
- Generate Memory Address
- Access data cache or calculate ALU result
- Write register result

Superscalar Execution

Can Execute Instructions I1 & I2 in Parallel if:
- Both are "simple" instructions
  - Don't require microcode sequencing
  - Some operations require U-pipe resources
  - 90% of SpecInt instructions
- I1 is not a jump
- Destination of I1 not source of I2
  - But can handle I1 setting CC and I2 being cond. jump
  - Destination of I1 not destination of I2

If Conditions Don't Hold
- Issue I1 to U Pipe
- I2 issued on next cycle
  - Possibly paired with following instruction

Branch Prediction

Branch Target Buffer
- Stores information about previously executed branches
  - Indexed by instruction address
  - Specifies branch destination + whether or not taken
- 256 entries

Branch Processing
- Look for instruction in BTB
- If found, start fetching at destination
- Branch condition resolved early in WB
  - If prediction correct, no branch penalty
  - If prediction incorrect, lose ~3 cycles
    » Which corresponds to > 3 instructions
- Update BTB

Superscalar Terminology

Basic
- Superscalar
  - Able to issue > 1 instruction / cycle
- Superpipelined
  - Deep, but not superscalar pipeline.
  - E.g., MIPS R5000 has 8 stages
Branch prediction
  - Logic to guess whether or not branch will be taken, and possibly branch target

Advanced
- Out-of-order
  - Able to issue instructions out of program order
- Speculation
  - Execute instructions beyond branch points, possibly nullifying later
- Register renaming
  - Able to dynamically assign physical registers to instructions
- Retire unit
  - Logic to keep track of instructions as they complete.
**Superscalar Execution Example**

**Assumptions**
- Single FP adder takes 2 cycles
- Single FP multiplier takes 5 cycles
- Can issue add & multiply together
- Must issue in-order (Single adder, data dependence)

**Data Flow**

\[ \text{Critical Path} = 9 \text{ cycles} \]

\[ \begin{align*}
  v &: \text{addt} \; \$f2, \; \$f4, \; \$f10 \\
  w &: \text{mult} \; \$f10, \; \$f6, \; \$f10 \\
  x &: \text{addt} \; \$f10, \; \$f8, \; \$f12 \\
  y &: \text{addt} \; \$f4, \; \$f6, \; \$f4 \\
  z &: \text{addt} \; \$f4, \; \$f8, \; \$f10 \\
\end{align*} \]

**Adding Advanced Features**

**Out Of Order Issue**
- Can start y as soon as adder available
- Must hold z until \$f10 not busy & adder available

\[ \begin{align*}
  v &: \text{addt} \; \$f2, \; \$f4, \; \$f10 \\
  w &: \text{mult} \; \$f10, \; \$f6, \; \$f10 \\
  x &: \text{addt} \; \$f10, \; \$f8, \; \$f12 \\
  y &: \text{addt} \; \$f4, \; \$f6, \; \$f4 \\
  z &: \text{addt} \; \$f4, \; \$f8, \; \$f10 \\
\end{align*} \]

**With Register Renaming**

\[ \begin{align*}
  v &: \text{addt} \; \$f2, \; \$f4, \; \$f10a \\
  w &: \text{mult} \; \$f10a, \; \$f6, \; \$f10a \\
  x &: \text{addt} \; \$f10a, \; \$f8, \; \$f12 \\
  y &: \text{addt} \; \$f4, \; \$f6, \; \$f4 \\
  z &: \text{addt} \; \$f4, \; \$f8, \; \$f10 \\
\end{align*} \]

**Pentium Pro (P6)**

**History**
- Announced in Feb. '95
- Delivering in high end machines now

**Features**
- Dynamically translates instructions to more regular format
  - Very wide RISC instructions
- Executes operations in parallel
  - Up to 5 at once
- Very deep pipeline
  - 12-18 cycle latency

**PentiumPro Block Diagram**
PentiumPro Operation

Translates instructions dynamically into “Uops”
- 118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine
- Uop executed when
  - Operands available
  - Functional unit available
- Execution controlled by “Reservation Stations”
  - Keeps track of data dependencies between uops
  - Allocates resources

Branch Prediction

Critical to Performance
- 11-15 cycle penalty for misprediction

Branch Target Buffer
- 512 entries
- 4 bits of history
- Adaptive algorithm
  - Can recognize repeated patterns, e.g., alternating taken-not taken

Handling BTB misses
- Detect in cycle 6
- Predict taken for negative offset, not taken for positive
  - Loops vs. conditionals

PPC 604

Superscalar
- Up to 4 instructions per cycle

Speculative & Out-of-Order Execution
- Begin issuing and executing instructions beyond branch

Other Processors in this Category
- MIPS R10000
- Intel PentiumPro & Pentium II
- Digital Alpha 21264

604 Block Diagram
General Principles

Must be Able to Flush Partially-Executed Instructions
- Branch mispredictions
- Earlier instruction generates exception

Special Treatment of "Architectural State"
- Programmer-visible registers
- Memory locations
- Don’t do actual update until certain instruction should be executed

Emulate "Data Flow" Execution Model
- Instruction can execute whenever operands available

Processing Stages

Fetch
- Get instruction from instruction cache

Dispatch (≈ Decode)
- Get available operands
- Assign to hardware execution unit

Execute
- Perform computation or memory operation
  - Store’s are only buffered

Retire / Commit (≈ Writeback)
- Allow architectural state to be updated
  - Register update
  - Buffered store

Fetching Instructions

- Up to 4 fetched from instruction cache in single cycle

Branch Target Address Cache (BTAC)
- Target addresses of recently-executed, predicted-taken branches
  - 64 entries
  - Indexed by instruction address
- Accessed in parallel with instruction fetch
- If hit, fetch at predicted target starting next cycle

Branch Prediction

Branch History Table (BHT)
- 512 state machines, indexed by low-order bits of instruction address
- Encode information about prior history of branch instructions
  - Small chance of two branch instructions aliasing
  - Predict whether or not branch will be taken
    - 3 cycle penalty if mispredict

Interaction with BTAC
- BHT entries start in state No!
- When make transition from No! to Yes?, allocate entry in BTAC
- Deallocate when make transition from Yes? to No!
Dispatching Actions

Generate Entry in Retirement Buffer
- 16-entry buffer tracking instructions currently "in flight"
  - Dispatched but not yet completed
  - Circular buffer in program order
  - Instruction tagged with branches they depend on
  - Easy to flush if mispredicted

Assign Rename Register as Target
- Additional registers (12 integer, 8 FP) used as targets for in-flight instructions
  - Instruction updates this register
  - Update of actual architectural register occurs only when instruction retired

Hazard Handling with Renaming

Dispatch Unit Maintains Mapping
- From register ID to actual register
- Could be the actual architectural register
  - Not target of currently-executing instruction
- Could be rename register
  - Perhaps already written by instruction that has not been retired
    - E.g., still waiting for confirmation of branch prediction
  - Perhaps instruction result not yet computed
    - Grab later when available

Hazards
- RAW: Mapping identifies operand source
- WAR: Write will be to different rename register
- WAW: Writes will be to different rename register

Read-after-Write (RAW) Dependences

Also known as a "true" dependence

Example:
S1:   addq r1, r2, r3
S2:   addq r3, r4, r4

How to optimize?
- cannot be optimized away
**Write-after-Read (WAR) Dependences**

Also known as an “anti” dependence

**Example:**

\[
\begin{align*}
S1: & \quad \text{addq } r1, r2, r3 \\
S2: & \quad \text{addq } r4, r5, r1 \\
\ldots & \\
    & \quad \text{addq } r1, r6, r7
\end{align*}
\]

**How to optimize?**

- rename dependent register (e.g., \( r1 \) in \( S2 \) \( \rightarrow r8 \))

\[
\begin{align*}
S1: & \quad \text{addq } r1, r2, r3 \\
S2: & \quad \text{addq } r4, r5, r8 \\
\ldots & \\
    & \quad \text{addq } r8, r6, r7
\end{align*}
\]

**Write-after-Write (WAW) Dependences**

Also known as an “output” dependence

**Example:**

\[
\begin{align*}
S1: & \quad \text{addq } r1, r2, r3 \\
S2: & \quad \text{addq } r4, r5, r3 \\
\ldots & \\
    & \quad \text{addq } r3, r6, r7
\end{align*}
\]

**How to optimize?**

- rename dependent register (e.g., \( r3 \) in \( S2 \) \( \rightarrow r8 \))

\[
\begin{align*}
S1: & \quad \text{addq } r1, r2, r3 \\
S2: & \quad \text{addq } r4, r5, r8 \\
\ldots & \\
    & \quad \text{addq } r8, r6, r7
\end{align*}
\]

**Moving Instructions Around**

**Reservation Stations**

- Buffers associated with execution units
- Hold instructions prior to execution
- May be waiting for one or more operands
- May be waiting for unit to be available

**Completion Busses**

- Results generated by execution units
- Tagged by rename register ID
- Monitored by reservation stations
- So they can get needed operands
- Effectively implements bypassing
- Supply results to completion unit

**Execution Resources**

**Integer**

- Two units to handle regular integer instructions
- One for “complex” operations
  - Multiply with latency 3--4 and throughput once per 1--2 cycles
  - Unpipelined divide with latency 20

**Floating Point**

- Add/multiply with latency 3 and throughput 1
- Unpipelined divide with latency 18--31

**Load Store Unit**

- Own address ALU
- Buffer of pending store instructions
  - Don’t perform actual store until ready to retire instruction
  - Loads can be performed speculatively
  - Check to see if target of pending store operation
Retiring Instructions

Retire in Program Order
- When instruction is at head of buffer
- Up to 4 per cycle
- Enable change of architectural state
  - Transfer from rename register to architectural
  - Free rename register for use by another instruction
  - Allow pending store operation to take place

Flush if Should not be Executed
- Tagged by branch that was mispredicted
- Follows instruction that raised exception
- As if instructions had never been fetched

Execution Example

Assumptions
- Two-way issue with renaming
- Rename registers $f0, f2, etc.
- 1 cycle add.d latency, 2 cycle mult.d

<table>
<thead>
<tr>
<th>Value</th>
<th>Rename</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f2</td>
<td>10.0</td>
<td>F</td>
</tr>
<tr>
<td>$f4</td>
<td>20.0</td>
<td>F</td>
</tr>
<tr>
<td>$f6</td>
<td>40.0</td>
<td>F</td>
</tr>
<tr>
<td>$f8</td>
<td>80.0</td>
<td>F</td>
</tr>
<tr>
<td>$f10</td>
<td>160.0</td>
<td>F</td>
</tr>
<tr>
<td>$f12</td>
<td>320.0</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Renames</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f0</td>
<td>--</td>
<td>F</td>
</tr>
<tr>
<td>$f2</td>
<td>--</td>
<td>F</td>
</tr>
<tr>
<td>$f4</td>
<td>--</td>
<td>F</td>
</tr>
<tr>
<td>$f6</td>
<td>--</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Terminals</th>
<th>ADB</th>
<th>MULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Dest</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Renames Dest</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Valid Dest</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Execution Example Cycle 1

Actions
- Instructions v & w issued
  - v target set to %f0
  - w target set to %f2

<table>
<thead>
<tr>
<th>Value</th>
<th>Rename</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f2</td>
<td>10.0</td>
<td>F</td>
</tr>
<tr>
<td>$f4</td>
<td>20.0</td>
<td>F</td>
</tr>
<tr>
<td>$f6</td>
<td>40.0</td>
<td>F</td>
</tr>
<tr>
<td>$f8</td>
<td>80.0</td>
<td>F</td>
</tr>
<tr>
<td>$f10</td>
<td>160.0</td>
<td>F</td>
</tr>
<tr>
<td>$f12</td>
<td>320.0</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Renames</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f0</td>
<td>--</td>
<td>F</td>
</tr>
<tr>
<td>$f2</td>
<td>--</td>
<td>F</td>
</tr>
<tr>
<td>$f4</td>
<td>--</td>
<td>F</td>
</tr>
<tr>
<td>$f6</td>
<td>--</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Terminals</th>
<th>ADB</th>
<th>MULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Dest</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Renames Dest</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Valid Dest</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Figure 3. The PowerPC 604 incorporates 8.6 million transistors on a 12.4 × 10.8 mm die using 0.66 micron, four-layer metal CMOS.
Execution Example Cycle 2

Actions
- Instructions x & y issued
  - x & y targets set to %f4 and %f6
- Instruction v executed
  - v:
    - addt $f2, $f4, $f10
- Instruction w begins execution
  - w:
    - mult $f10, $f6, $f10
- Instruction y executed
  - y:
    - addt $f4, $f6, $f4
- Instruction z stalled
  - Not enough reservation stations
  - Assigned to %f0

Execution Example Cycle 3

- Instruction v retired
  - But doesn't change $f10
- Instruction w begins execution
  - w:
    - mult $f10, $f6, $f10
- Instruction y executed
  - y:
    - addt $f4, $f6, $f4
- Instruction z stalled
  - Not enough reservation stations
  - Assigned to %f0

Execution Example Cycle 4

- Instruction w finishes execution
- Instruction y cannot be retired yet
- Instruction z issued
  - Assigned to %f0

Execution Example Cycle 5

- Instruction w retired
  - But does not change $f10
- Instruction x cannot be retired yet
- Instruction x executed
Execution Example Cycle 6

- Instruction x & y retired
  - Update $f12$ and $f4$
- Instruction z executed

<table>
<thead>
<tr>
<th>Value</th>
<th>Rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f2$</td>
<td>10.0</td>
</tr>
<tr>
<td>$f4$</td>
<td>80.0</td>
</tr>
<tr>
<td>$f6$</td>
<td>40.0</td>
</tr>
<tr>
<td>$f8$</td>
<td>80.0</td>
</tr>
<tr>
<td>$f10$</td>
<td>160.0</td>
</tr>
<tr>
<td>$f12$</td>
<td>200.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op1</th>
<th>Op2</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td></td>
<td>ADD</td>
</tr>
</tbody>
</table>

\[ v: \text{addt} \; f2, \; f4, \; f10 \]
\[ w: \text{mult} \; f10, \; f6, \; f10 \]
\[ x: \text{addt} \; f2, \; f4, \; f10 \]
\[ y: \text{addt} \; f2, \; f4, \; f4 \]
\[ z: \text{addt} \; f4, \; f8, \; f10 \]

Value Rename

- 10.0 $f2$
- 60.0 $f4$
- 40.0 $f6$
- 80.0 $f8$
- 160.0 $f10$
- 200.0 $f12$
- 140.0 $f10$

Living with Expensive Branches

Mispredicted Branch Carries a High Cost
- Must flush many in-flight instructions
- Start fetching at correct target
- Will get worse with deeper and wider pipelines

Impact on Programmer / Compiler
- Avoid conditionals when possible
  - Bit manipulation tricks
- Use special conditional-move instructions
  - Recent additions to many instruction sets
- Make branches predictable
  - Very low overhead when predicted correctly

Branch Prediction Example

\[ \text{static void loop1()} \{ \]
\[ \quad \text{int i;} \]
\[ \quad \text{data_t abs_sum = (data_t) 0;} \]
\[ \quad \text{data_t prod = (data_t) 1;} \]
\[ \quad \text{for (i = 0; i < CNT; i++) { \}
\[ \quad \quad \text{data_t x = data[i];} \]
\[ \quad \quad \text{int ax; ax = ABS(x);} \]
\[ \quad \quad \text{abs_sum += ax;} \]
\[ \quad \quad \text{prod *= x;} \]
\[ \quad \} \]
\[ \quad \text{answer = abs_sum*prod;} \]
\[ \} \]

MIPS Code

\[ \text{#define ABS(x) x < 0 ? -x : x} \]
\[ \text{static void loop1()} \{ \]
\[ \quad \text{int i;} \]
\[ \quad \text{data_t abs_sum = (data_t) 0;} \]
\[ \quad \text{data_t prod = (data_t) 1;} \]
\[ \quad \text{for (i = 0; i < CNT; i++) { \}
\[ \quad \quad \text{data_t x = data[i];} \]
\[ \quad \quad \text{int ax; ax = ABS(x);} \]
\[ \quad \quad \text{abs_sum += ax;} \]
\[ \quad \quad \text{prod *= x;} \]
\[ \quad \} \]
\[ \quad \text{answer = abs_sum*prod;} \]
\[ \} \]

- 10.0 $f2$
- 80.0 $f4$
- 40.0 $f6$
- 80.0 $f8$
- 140.0 $f10$
- 320.0 $f12$

- 140.0 $f10$
- 320.0 $f12$

\[ \text{ADD} \]

\[ \text{MULT} \]

\[ \text{ADD} \]

\[ \text{MULT} \]
Some Interesting Patterns

- PPPPPPPPP
  - Should give perfect prediction
- RRRRRRRRR
  - Will mispredict 1/2 of the time
- N*N[PNPN]
  - Should alternate between states No! and No?
- N*N[PPNN]
  - Should alternate between states No? and Yes?
- N*P[PNPN]
- N*P[PPNN]

Loop Performance (FP)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>RS3000 Cycles</th>
<th>PPC 604 Penalty</th>
<th>Pentium Cycles</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPPPPPPP</td>
<td>13.6</td>
<td>0</td>
<td>21.1</td>
<td>0</td>
</tr>
<tr>
<td>RRRRRRRR</td>
<td>13.6</td>
<td>0</td>
<td>23.9</td>
<td>1.8</td>
</tr>
<tr>
<td>N*N[PNPN]</td>
<td>13.6</td>
<td>0</td>
<td>25.3</td>
<td>2.2</td>
</tr>
<tr>
<td>N*P[PNPN]</td>
<td>13.3</td>
<td>0.3</td>
<td>24.3</td>
<td>2.2</td>
</tr>
<tr>
<td>N*N[PPNN]</td>
<td>13.3</td>
<td>-0.3</td>
<td>23.9</td>
<td>2.2</td>
</tr>
<tr>
<td>N*P[PPNN]</td>
<td>13.5</td>
<td>0</td>
<td>24.1</td>
<td>2.6</td>
</tr>
</tbody>
</table>

Observations
- 604 has prediction rates 0%, 50%, and 100%
  - Expected 50% from N*N[PNPN]
  - Expected 25% from N*N[PPNN]
- Loop so tight that speculate through single branch twice?
- Pentium appears to be more variable, ranging 0 to 100%

Special Patterns Can Be Worse than Random
- Only 50% of all people are "above average"

Pentium II
- Random shows clear penalty
- But others do well
  - More clever prediction algorithm

R10000
- Has special "conditional move" instructions
- Compiler translates `a = Cond ? Texpr : Fexpr` into
  `a = Fexpr`
  `temp = Texpr`
  `CMOV(a, temp, Cond)`
- Only valid if `Texpr & Fexpr` can’t cause error

P6 Branch Prediction

Two-Level Scheme
- Yeh & Patt, TSCA ’93
- Keep shift register showing past k outcomes for branch
- Use to index 2^k entry table
- Each entry provides 2-bit, saturating counter predictor
- Very effective for any deterministic branching pattern
Branch Prediction Comparisons

DEC Alpha 21264

Properties:
- 4-6 way superscalar
- Out of order execution with renaming
- Up to 80 instructions in process simultaneously
- Lots of cache & memory bandwidth

DEC Alpha 21264 Block Diagram

4 Integer ALUs
- Each can perform simple instructions
- 2 handle address calculations

Register Files
- 32 arch / 80 physical Int
- 32 arch / 72 physical FP
- Int registers duplicated
  - Extra cycle delay from write in one to read in other
  - Each has 6 read ports, 4 write ports
  - Attempt to issue consumer to producer side

DEC Alpha 21264 Pipeline

Very Deep Pipeline
- Can’t do much in 2ns clock cycle!
- 7 cycles for simple instruction
- 9 cycles for load or store
- 7 cycle penalty for mispredicted branch
- Elaborate branch predication logic
- Claim 95% accuracy

Microprocessor Report 10/28/96

Microprocessor Report 10/28/96
21264 Branch Prediction Logic

- Purpose: Predict whether or not branch taken
- 35KB of prediction information
- 2% of total die size
- Claim 0.7--1.0% misprediction

Challenges Ahead

- Diminishing Returns on Cost vs. Performance
  - Superscalar processors require instruction level parallelism
  - Many programs limited by sequential dependencies

- Finding New Sources of Parallelism
  - e.g., thread-level parallelism

- Getting Design Correct Difficult
  - Verification team larger than design team
  - Devise tests for interactions between concurrent instructions
    - May be 80 executing at once

New Era for Performance Optimization

- Data Resources are Free and Fast
  - Plenty of computational units
  - Most programs have poor utilization

- Unexpected Changes in Control Flow Expensive
  - Kill everything downstream when mispredict
  - Even if will execute in near future where branches reconverge

- Think Parallel
  - Try to get lots of things going at once

- Not a Truly Parallel Machine
  - Bounded resources
  - Access from limited code window