This Unit: Superscalar Execution

- Idea of instruction-level parallelism
- Superscalar scaling issues
  - Multiple fetch and branch prediction
  - Dependence-checks & stall logic
  - Wide bypassing
  - Register file & cache bandwidth
- “Superscalar” vs VLIW/EPIC

A Key Theme of CIS 501: Parallelism

- Previously: pipeline-level parallelism
  - Work on execute of one instruction in parallel with decode of next
- Next: instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel
  - Today: multiple issue
- Later:
  - Static & dynamic scheduling
    - Extract much more ILP
  - Data-level parallelism (DLP)
    - Single-instruction, multiple data (one insn., four 64-bit adds)
  - Thread-level parallelism (TLP)
    - Multiple software threads running on multiple cores

Readings

- Textbook (MA:FSPTCM)
  - Sections 3.1, 3.2 (but not “Sidebar” in 3.2), 3.5.1
  - Sections 4.2, 4.3, 5.3.3

A Key Theme of CIS 501: Parallelism

- Previously: pipeline-level parallelism
  - Work on execute of one instruction in parallel with decode of next
- Next: instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel
  - Today: multiple issue
- Later:
  - Static & dynamic scheduling
    - Extract much more ILP
  - Data-level parallelism (DLP)
    - Single-instruction, multiple data (one insn., four 64-bit adds)
  - Thread-level parallelism (TLP)
    - Multiple software threads running on multiple cores

A Key Theme of CIS 501: Parallelism

- Previously: pipeline-level parallelism
  - Work on execute of one instruction in parallel with decode of next
- Next: instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel
  - Today: multiple issue
- Later:
  - Static & dynamic scheduling
    - Extract much more ILP
  - Data-level parallelism (DLP)
    - Single-instruction, multiple data (one insn., four 64-bit adds)
  - Thread-level parallelism (TLP)
    - Multiple software threads running on multiple cores

A Key Theme of CIS 501: Parallelism

- Previously: pipeline-level parallelism
  - Work on execute of one instruction in parallel with decode of next
- Next: instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel
  - Today: multiple issue
- Later:
  - Static & dynamic scheduling
    - Extract much more ILP
  - Data-level parallelism (DLP)
    - Single-instruction, multiple data (one insn., four 64-bit adds)
  - Thread-level parallelism (TLP)
    - Multiple software threads running on multiple cores

A Key Theme of CIS 501: Parallelism

- Previously: pipeline-level parallelism
  - Work on execute of one instruction in parallel with decode of next
- Next: instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel
  - Today: multiple issue
- Later:
  - Static & dynamic scheduling
    - Extract much more ILP
  - Data-level parallelism (DLP)
    - Single-instruction, multiple data (one insn., four 64-bit adds)
  - Thread-level parallelism (TLP)
    - Multiple software threads running on multiple cores

A Key Theme of CIS 501: Parallelism

- Previously: pipeline-level parallelism
  - Work on execute of one instruction in parallel with decode of next
- Next: instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel
  - Today: multiple issue
- Later:
  - Static & dynamic scheduling
    - Extract much more ILP
  - Data-level parallelism (DLP)
    - Single-instruction, multiple data (one insn., four 64-bit adds)
  - Thread-level parallelism (TLP)
    - Multiple software threads running on multiple cores

A Key Theme of CIS 501: Parallelism

- Previously: pipeline-level parallelism
  - Work on execute of one instruction in parallel with decode of next
- Next: instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel
  - Today: multiple issue
- Later:
  - Static & dynamic scheduling
    - Extract much more ILP
  - Data-level parallelism (DLP)
    - Single-instruction, multiple data (one insn., four 64-bit adds)
  - Thread-level parallelism (TLP)
    - Multiple software threads running on multiple cores
Scalar Pipeline and the Flynn Bottleneck

- So far we have looked at **scalar pipelines**
  - One instruction per stage
    - With control speculation, bypassing, etc.
    - Performance limit (aka "Flynn Bottleneck") is CPI = IPC = 1
    - Limit is never even achieved (hazards)
    - Diminishing returns from "super-pipelining" (hazards + overhead)

Multiple-Issue Pipeline

- Overcome this limit using **multiple issue**
  - Also called **superscalar**
  - Two instructions per stage at once, or three, or four, or eight...
  - "**Instruction-Level Parallelism (ILP)**" [Fisher, IEEE TC'81]
- Today, typically “4-wide” (Intel Core i7, AMD Opteron)
  - Some more (Power5 is 5-issue; Itanium is 6-issue)
  - Some less (dual-issue is common for simple cores)

A Typical Dual-Issue Pipeline

- Fetch an entire 16B or 32B cache block
  - 4 to 8 instructions (assuming 4-byte average instruction length)
  - Predict a single branch per cycle
- Parallel decode
  - Need to check for conflicting instructions
  - Output of $I_1$ is an input to $I_2$
  - Other stalls, too (for example, load-use delay)

A Typical Dual-Issue Pipeline

- Multi-ported register file
  - Larger area, latency, power, cost, complexity
- Multiple execution units
  - Simple adders are easy, but bypass paths are expensive
- Memory unit
  - Single load per cycle (stall at decode) probably okay for dual issue
  - Alternative: add a read port to data cache
    - Larger area, latency, power, cost, complexity
Superscalar Pipeline Diagrams - Ideal

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 0(r1)</td>
<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>
<tr>
<td>lw 4(r1)</td>
<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>
<tr>
<td>lw 8(r1)</td>
<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>
<tr>
<td>add r14,r15</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r12,r13</td>
<td>r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r17,r16</td>
<td>r8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 0(r18)</td>
<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>

2-way superscalar

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 0(r1)</td>
<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>
<tr>
<td>lw 4(r1)</td>
<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>
<tr>
<td>lw 8(r1)</td>
<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>
<tr>
<td>add r14,r15</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r12,r13</td>
<td>r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r17,r16</td>
<td>r8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 0(r18)</td>
<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>

Superscalar Challenges - Front End

- **Superscalar instruction fetch**
  - Modest: need multiple instructions per cycle
  - Aggressive: predict multiple branches

- **Superscalar instruction decode**
  - Replicate decoders

- **Superscalar instruction issue**
  - Determine when instructions can proceed in parallel
  - Not all combinations possible
  - More complex stall logic - order $N^2$ for $N$-wide machine

- **Superscalar register read**
  - One port for each register read
  - Each port needs its own set of address and data wires
  - Example, 4-wide superscalar $\rightarrow$ 8 read ports

Superscalar Pipeline Diagrams - Realistic

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 0(r1)</td>
<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>
<tr>
<td>lw 4(r1)</td>
<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>
<tr>
<td>lw 8(r1)</td>
<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>
<tr>
<td>add r4,r5</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2,r3</td>
<td>r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r7,r6</td>
<td>r8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 4(r8)</td>
<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>

2-way superscalar

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 0(r1)</td>
<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>
<tr>
<td>lw 4(r1)</td>
<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>
<tr>
<td>lw 8(r1)</td>
<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>
<tr>
<td>add r4,r5</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2,r3</td>
<td>r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r7,r6</td>
<td>r8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 4(r8)</td>
<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>

Superscalar Challenges - Back End

- **Superscalar instruction execution**
  - Replicate arithmetic units
  - Perhaps multiple cache ports

- **Superscalar bypass paths**
  - More possible sources for data values
  - Order $(N^2 \times P)$ for $N$-wide machine with execute pipeline depth $P$

- **Superscalar instruction register writeback**
  - One write port per instruction that writes a register
  - Example, 4-wide superscalar $\rightarrow$ 4 write ports

- **Fundamental challenge:**
  - Amount of ILP (instruction-level parallelism) in the program
  - Compiler must schedule code and extract parallelism
How Much ILP is There?

• The compiler tries to “schedule” code to avoid stalls
  • Even for scalar machines (to fill load-use delay slot)
  • Even harder to schedule multiple-issue (superscalar)
• How much ILP is common?
  • Greatly depends on the application
    • Consider memory copy
    • Unroll loop, lots of independent operations
  • Other programs, less so
• Even given unbounded ILP, superscalar has implementation limits
  • IPC (or CPI) vs clock frequency trade-off
  • Given these challenges, what is reasonable today?
    • ~4 instruction per cycle maximum

Superscalar Execution

Superscalar Decode & Register Read

• What is involved in decoding multiple (N) insns per cycle?
• Actually doing the decoding?
  • Easy if fixed length (multiple decoders), doable if variable length
• Reading input registers?
  • Nominally, 2N read + N write (2 read + 1 write per insn)
    – Latency, area $\propto$ #ports$^2$
• What about the stall logic?

N$^2$ Dependence Cross-Check

• Stall logic for 1-wide pipeline with full bypassing
  • Full bypassing $\rightarrow$ load/use stalls only
    $X/M$.op==LOAD && (D/X.rs1==X/M.rd || D/X.rs2==X/M.rd)
  • Two “terms”: $\propto$ 2N
• Now: same logic for a 2-wide pipeline
  $X/M_1$.op==LOAD && (D/X_1.rs1==X/M_1.rd || D/X_1.rs2==X/M_1.rd) ||
  $X/M_1$.op==LOAD && (D/X_1.rs1==X/M_1.rd || D/X_2.rs2==X/M_1.rd) ||
  $X/M_2$.op==LOAD && (D/X_1.rs1==X/M_2.rd || D/X_1.rs2==X/M_2.rd) ||
  $X/M_2$.op==LOAD && (D/X_2.rs1==X/M_2.rd || D/X_2.rs2==X/M_2.rd)
  • Eight “terms”: $\propto$ 2N$^2$
  • N$^2$ dependence cross-check
  • Not quite done, also need
    $D/X_2.rs1==D/X_1.rd || D/X_2.rs2==D/X_1.rd$
Superscalar Execute

- What is involved in executing N insns per cycle?
- Multiple execution units ... N of every kind?
  - N ALUs? OK, ALUs are small
  - N floating point dividers? No, dividers are big, \$fdiv\$ is uncommon
  - How many branches per cycle? How many loads/stores per cycle?
  - Typically some mix of functional units proportional to insn mix
    - Intel Pentium: 1 any + 1 "simple" (such as ADD, etc.)
    - Alpha 21164: 2 integer (including 2 loads) + 2 floating point

Superscalar Memory Access

- What about multiple loads/stores per cycle?
  - Probably only necessary on processors 4-wide or wider
    - Core i7: is one load & one store per cycle
  - More important to support multiple loads than multiple stores
    -Insn mix: loads (~20–25%), stores (~10–15%)
    -Alpha 21164: two loads or one store per cycle

D$ Bandwidth: Multi-Porting, Replication

- How to provide additional D$ bandwidth?
  - Have already seen split I$/D$, but that gives you just one D$ port
  - How to provide a second (maybe even a third) D$ port?

- Option #1: multi-porting
  + Lots of wires; expensive in terms of latency, area (cost), and power

- Option #2: replication
  + Read from either replica, but writes update both replicas
    - Writing both insures they have the same values
  + General solution for loads, little latency penalty
    - Not a solution for stores (that’s OK), area (cost), power penalty

D$ Bandwidth: Banking

- Option #3: banking (or interleaving)
  - Divide D$ into “banks” (by address), one access per bank per cycle
    - Bank conflict: two accesses to same bank → one stalls
      - No latency, area, power overheads (latency may even be lower)
      - One access per bank per cycle, assuming no conflicts
        - Complex stall logic → address not known until execute stage
        - To support N accesses, need 2N+ banks to avoid frequent conflicts

- Which address bit(s) determine bank?
  - Offset bits? Individual cache lines spread among different banks
    - Fewer conflicts
      - Must replicate tags across banks, complex miss handling
  - Index bits? Banks contain complete cache lines
    - More conflicts
      - Tags not replicated, simpler miss handling
Superscalar Register Read/Write

- How many register file ports to execute N insns per cycle?
  - Nominally, 2N read + N write (2 read + 1 write per insn)
    - Latency, area $\propto$ #ports$^2$
  - In reality, fewer than that
    - Read ports: some instructions read only one register
    - Write ports: stores, branches (35% insns) don't write registers
- Multi-porting and replication both work for register files
  - Alpha 21264 used replication (more in a bit)
- Banking? Not used (conflicts too hard to handle)

Superscalar Bypass

- $N^2$ bypass network
  - N+1 input muxes at each ALU input
  - $N^2$ point-to-point connections
  - Routing lengthens wires
  - Heavy capacitive load
    - And this is just one bypass stage (MX)!
    - There is also WX bypassing
    - Even more for deeper pipelines
- One of the big problems of superscalar

Not All $N^2$ Created Equal

- $N^2$ bypass vs. $N^2$ stall logic & dependence cross-check
  - Which is the bigger problem?
- $N^2$ bypass ... by far
  - 64-bit quantities (vs. 5-bit)
  - Multiple levels (MX, WX) of bypass (vs. 1 level of stall logic)
  - Must fit in one clock period with ALU (vs. not)
- Dependence cross-check not even 2nd biggest $N^2$ problem
  - Regfile is also an $N^2$ problem (think latency where N is #ports)
  - And also more serious than cross-check

Mitigating $N^2$ Bypass: Clustering

- **Clustering**: mitigates $N^2$ bypass
  - Group ALUs into K clusters
  - Full bypassing within a cluster
  - Limited bypassing between clusters
    - With 1 or 2 cycle delay
      - $(N/K) + 1$ inputs at each mux
      - $(N/K)^2$ bypass paths in each cluster
  - **Steering**: key to performance
    - Steer dependent insns to same cluster
    - Statically (compiler) or dynamically
    - Hurts IPC, allows wide issue at same clock
    - E.g., Alpha 21264
      - Bypass wouldn't fit into clock cycle
      - 4-wide, 2 clusters
Mitigating $N^2$ RegFile: Clustering++

- **Clustering**: split $N$-wide execution pipeline into $K$ clusters
  - With centralized register file, $2N$ read ports and $N$ write ports

- **Clustered register file**: extend clustering to register file
  - Replicate the register file (one replica per cluster)
  - Register file supplies register operands to just its cluster
  - All register writes go to all register files (keep them in sync)
  - Advantage: fewer read ports per register!
  - $K$ register files, each with $2N/K$ read ports and $N$ write ports

Simple Superscalar Fetch

- What is involved in fetching multiple instructions per cycle?
- In same cache block? → no problem
  - 64-byte cache block is 16 instructions (~4 bytes per instruction)
  - Favors larger block size (independent of hit rate)
- What if next instruction is last instruction in a block?
  - Fetch only one instruction that cycle
  - Or, some processors may allow fetching from 2 consecutive blocks
- Compilers align code to I$ blocks (.align directive in asm)
  - Reduces I$ capacity
  - Increases fetch bandwidth utilization (more important)
Limits of Simple Superscalar Fetch

- How many instructions can be fetched on average?
  - BTB predicts the next block of instructions to fetch
  - Support multiple branch (direction) predictions per cycle
  - Discard post-branch insns after first branch predicted as "taken"
  - Lowers effective fetch width and IPC
  - Average number of instructions per taken branch?
    - Assume: 20% branches, 50% taken → ~10 instructions
- Consider a 5-instruction loop with an 4-issue processor
  - Without smarter fetch, ILP is limited to 2.5 (not 4)
  - Compiler could "unroll" the loop (reduce taken branches)
- How else can we increase fetch rate?

Increasing Superscalar Fetch Rate

- Option #1: over-fetch and buffer
  - Add a queue between fetch and decode (18 entries in Intel Core2)
  - Compensates for cycles that fetch less than maximum instructions
  - "decouples" the "front end" (fetch) from the "back end" (execute)
  - Option #2: predict next two blocks (extend BTB)
    - Transmits two PCs to fetch stage: "next PC" and "next-next PC"
    - Access I-cache twice (requires multiple ports or banks)
    - Requires extra merging logic to select and merge correct insns
      - Elongates pipeline, increases branch penalty

Impact of Branch Prediction

- Base CPI for scalar pipeline is 1
- **Base CPI for N-way superscalar pipeline is 1/N**
  - Amplifies stall penalties
  - Assumes no data stalls (an overly optimistic assumption)
- Example: Branch penalty calculation
  - 20% branches, 75% taken, 2 cycle penalty, no branch prediction
  - Scalar pipeline
    - $1 + 0.2 \times 0.75 \times 2 = 1.3 \rightarrow 1.3/1 = 1.3 \rightarrow 30\%$ slowdown
  - 2-way superscalar pipeline
    - $0.5 + 0.2 \times 0.75 \times 2 = 0.8 \rightarrow 0.8/0.5 = 1.6 \rightarrow 60\%$ slowdown
  - 4-way superscalar
    - $0.25 + 0.2 \times 0.75 \times 2 = 0.55 \rightarrow 0.55/0.25 = 2.2 \rightarrow 120\%$ slowdown
Predication (not prediction, predication)

- Branch mis-predictions hurt more on superscalar
  - Replace difficult branches with something else...
  - Convert control flow into data flow (& dependencies)
  - Avoids mis-predictions by removing hard-to-predict branches
  - Can hurt performance if branch was highly predictable
- **Predication**: insns conditionally executed
  - **Full predication** (ARM, Intel Itanium)
    - Can tag every insn with predicate, but extra bits in instruction
  - **Conditional moves** (Alpha, x86)
    - Construct appearance of full predication from one primitive
      - `cmoveq r1, r2, r3` // if (r1==0) r3=r2;
      - Only good way of adding predication to an existing ISA

- **If-conversion**: replacing control with predication

ISA Support for Predication

- Itanium: change branch 1 to **set-predicate insn** `fspne`
- Change insns 2 and 4 to **predicated insns**
  - `ldf.p` performs `ldf` if predicate `pl` is true
  - `stf.np` performs `stf` if predicate `pl` is false

Predication If-Conversion Example

Source code

```
A = Y[i];
if (A == 0)
   A = W[i];
else
   Y[i] = 0;
Z[i] = A*X[i];
```

Machine code

```
0: ldf Y(r1), f2
1: fbne f2, 4
2: ldf W(r1), f2
3: jump 5
4: stf f0, Y(r1)
5: ldf X(r1), f4
6: mulf f4, f2, f6
7: stf f6, Z(r1)
```

Using Predication

```
0: ldf Y(r1), f2
1: fspne f2, p1
2: ldf.p p1, W(r1), f2
4: stf.np p1, f0, Y(r1)
5: ldf X(r1), f4
6: mulf f4, f2, f6
7: stf f6, Z(r1)
```

CMOV Prediction Example

```
int func(int a, int b, int* array)
{
    if (a > 0) {
        return b;
    } else {
        return array[b];
    }
}
```

```
int func2(int a, int b, int* array)
{
    int temp = array[b];
    if (a > 0) {
        return b;
    } else {
        return temp;
    }
}
```

```
func:  testl %edi, %edi
       jg .L2
       movslq %esi, %rax
       movslq %esi, %eax
       movl (%rdx, %rax, 4), %esi
       movl (%rdx, %rax, 4), %esi
       ret .L2:
```

- x86 only has a “CMOV” instruction
  - Note: in x86’s CMOV, any “load” part is non-conditional
  - Small change in the code helps the compiler optimize
Another CMOV Example (Part I)

- gcc --os --no-if-conversion

```c
tree_t* search(tree_t* t, int key) {
    while (t != NULL) {
        if (t->value == key) {
            return t;
        } else {
            t = t->left_ptr;
            if (t->value > key) {
                t = t->right_ptr;
            } else {
                t = t->left_ptr;
            }
        }
    }
    return NULL;
}
```

- Baseline
  - Same with and without --no-if-conversion flag!

Another CMOV Example (Part II)

- gcc --os --no-if-conversion

```c
tree_t* search(tree_t* t, int key) {
    cmpl %esi, (%rdi)
    je L4
    jle L6
    movq 8(%rdi), %rdi
    jmp L12
    L6:
    movq 16(%rdi), %rdi
    L12:
    testq %rdi, %rdi
    jne L3
}
```

- Similar assembly as before (-no-if-conversion)
  - Does reduce taken branches

Another CMOV Example (Part III)

- gcc --os

```c
tree_t* search(tree_t* t, int key) {
    cmpl %esi, (%rdi)
    je L4
    movq 8(%rdi), %rax
    movq 16(%rdi), %rdi
    jmp L12
    L12:
    testq %rdi, %rdi
    jne L3
}
```

- Now, with --if-conversion (enabled by default)
  - Uses CMOV to avoid branch misprediction

Predication Performance

- Cost/benefit analysis
  - Benefit: predication avoids branches
    - Thus avoiding mis-predictions
    - Also reduces pressure on predictor table (fewer branches to track)
  - Cost: extra instructions (fetched, but not actually executed)
    - As branch predictors are highly accurate...
      - Might not help:
        - 5-stage pipeline, two instruction on each path of if-then-else
        - No performance gain, likely slower if branch predictable
      - Or even hurt!
    - But can help:
      - Deeper pipelines, hard-to-predict branches, and few added insn
  - Thus, prediction is useful, but not a panacea
Multiple Issue Implementations

- **Statically-scheduled (in-order) superscalar**
  - What we’ve talked about thus far
    + Executes unmodified sequential programs
    - Hardware must figure out what can be done in parallel
    + E.g., Pentium (2-wide), UltraSPARC (4-wide), Alpha 21164 (4-wide)

- **Very Long Instruction Word (VLIW)**
  - Compiler identifies independent instructions, new ISA
    + Hardware can be dumb and low power
    - E.g., TransMeta Crusoe (4-wide)
  - Variant: Explicitly Parallel Instruction Computing (EPIC)
    - A compromise: compiler does some, hardware does the rest
    + E.g., Intel Itanium (6-wide)

- **Dynamically-scheduled superscalar**
  - Hardware extracts more ILP by on-the-fly reordering
    + Core 2, Core i7 (4-wide), Alpha 21264 (4-wide)

Very Long Instruction Word (VLIW)

- Hardware-centric multiple issue problems
  - Wide fetch/branch prediction, \( N^2 \) bypass, \( N^2 \) dependence checks
  - Hardware solutions have been proposed: clustering, etc.

- **Compiler-centric: very long insn word (VLIW)**
  - Effectively, a 1-wide pipeline, but unit is an \( N \)-insn group
    + Started with “horizontal microcode”
  - Compiler ensures insns within a group are independent
    - If no independent insns, slots filled with \texttt{nops}
  - Group travels down pipeline as a unit
    + Simplifies pipeline control
    + Cross-checks within a group unnecessary
  - Downstream cross-checks still necessary
  - Typically “slotted”: 1st insn must be ALU, 2nd mem, etc.
    + Further simplification

VLIW Advantages

+ Simpler instruction fetch
  - Fetch a bundle per cycle
+ Simpler dependence check logic
  - Compiler guarantees all instructions in bundle independent
+ Simpler branch prediction
  - Restrict to one branch per bundle
+ By default, doesn’t help bypasses or register file problems
  - Which are the much bigger problems!
    + Although clustering and replication can help VLIW, too
  + Compiler-visible clustering possible in VLIW
    - Each “lane” of VLIW has “local” registers (read/written by this lane)
    - A few “global” registers (read/written by any lane) are used to communicate between lanes
VLIW Disadvantages

- Code density
  - Lots of "no-ops" in bundles
- Not compatible across machines of different widths
  - "not compatible" could mean programs would execute incorrectly
  - Or, "not compatible" can mean programs would execute slowly
  - Is non-compatibility worth all of this?
- How did TransMeta deal with compatibility problem?
  - Dynamically translates x86 to internal VLIW
  - GPUs also use VLIW, do dynamic translation of graphics operations
- Finally, VLIW doesn’t solve all problems
  - VLIW mainly targets dependence checking
  - Which isn’t the worst $N^2$ problem in multiple-issue
  - Doesn’t magically create ILP

EPIC

- **EPIC (Explicitly Parallel Insn Computing)**
  - Variant of VLIW (Variable Length Insn Words)
  - Implemented as "bundles" with explicit dependence bits
  - Helps code density
  - Code is compatible with different "bundle" width machines
  - E.g., Intel Itanium (IA-64)
  - 128-bit bundles (three 41-bit insns + 4 dependence bits)
  - Still does not address bypassing or register file issues

Trends in Single-Processor Multiple Issue

<table>
<thead>
<tr>
<th>Year</th>
<th>486</th>
<th>Pentium</th>
<th>PentiumII</th>
<th>Pentium4</th>
<th>Itanium</th>
<th>ItaniumII</th>
<th>Core2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

- Issue width has saturated at 4-6 for high-performance cores
  - Canceled Alpha 21464 was 8-way issue
  - Not enough ILP to justify going to wider issue
  - Hardware or compiler scheduling needed to exploit 4-6 effectively

- For high-performance *per watt* cores, issue width is ~2
  - Advanced scheduling techniques not as critical
  - Multi-threading (a little later) helps cope with cache misses
Next Up...

- Extracting more ILP via:
  - Static scheduling in the compiler
  - Dynamic scheduling in hardware

Multiple Issue Summary

- Superscalar hardware issues
  - Bypassing and register file
  - Stall logic
  - Fetch

- Multiple-issue designs
  - "Superscalar"
  - VLIW