Improving Performance

• Goal:
  - decrease CPI
  - increase clock

• Where are the bottlenecks?

• Where are the bottlenecks?
  - At most 1 instruction retired per cycle
  - Cycle time
  - hazards/bypass
    » data
    » control
    » ld/st
  - memory latency

• Possible solutions?
Possible approaches

- Superpipelining?
- Fast clocks?
- Data speculation?
- Control speculation?
- Id/st units?
- Caches?
- exploit ILP?
  - superscalar?
  - VLIW?
  - vector?
- Multithreading?

Keep in mind:
- power
- area
- typical program structure
  - BB size ~3-6
- Dependencies

"Scalar" Pipeline & 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)

Pipeline Depth

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

integer pipeline

floating point pipeline

An Opportunity...

But consider:
ADD r1, r2 -> r3
ADD r4, r5 -> r6
- Why not execute them at the same time? (We can!)

What about:
ADD r1, r2 -> r3
ADD r4, r3 -> r6
- In this case, dependences prevent parallel execution

What about three instructions at a time?
- Or four instructions at a time?
What Checking Is Required?

For two instructions: 2 checks
ADD src11, src21 -> dest1
ADD src12, src22 -> dest2 (2 checks)

For three instructions: 6 checks
ADD src11, src21 -> dest1
ADD src12, src22 -> dest2 (2 checks)
ADD src13, src23 -> dest3 (4 checks)

For four instructions: 12 checks
ADD src11, src21 -> dest1
ADD src12, src22 -> dest2 (2 checks)
ADD src13, src23 -> dest3 (4 checks)
ADD src14, src24 -> dest4 (6 checks)

Plus checking for load-to-use stalls from prior n loads

How do we build such “superscalar” hardware?

Multiple-Issue or “Superscalar” 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 (1 of 2)

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
  - Is output register of $I_1$ is an input register to $I_2$?
- Other stalls, too (for example, load-use delay)

A Typical Dual-Issue Pipeline (2 of 2)

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 Challenges - Front End

Superscalar instruction fetch
- Modest: fetch multiple instructions per cycle
- Aggressive: buffer instructions and/or predict multiple branches

Superscalar instruction decode
- Replicate decoders

Superscalar instruction issue
- Determine when instructions can proceed in parallel
- More complex stall logic - $O(N^2)$ for $N$-wide machine
- Not all combinations of types of instructions possible

Superscalar register read
- Port for each register read (4-wide superscalar $\Rightarrow$ 8 read "ports")
- Each port needs its own set of address and data wires
  - Latency & area $\propto$ #ports$^2$
Superscalar Challenges - Back End

Superscalar instruction execution
- Replicate arithmetic units (but not all, say, integer divider)
- Perhaps multiple cache ports (slower access, higher energy)
  - Only for 4-wide or larger (why? only ~35% are load/store instr)

Superscalar bypass paths
- More possible sources for data values
- $O(PN^2)$ 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

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 & Register File

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
  - Can hurt IPC, but faster clock
- $(N/K) + 1$ inputs at each mux
- $(N/K)^2$ bypass paths in each cluster

Steering: key to performance
- Steer dependent instructions to same cluster

Cluster register file, too
- Replicate a register file per cluster
- All register writes update all replicas
- Fewer read ports; only for cluster
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

Another Challenge: Superscalar Fetch

What is involved in fetching multiple instrns 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

What about taken branches?
- How many instructions can be fetched on average?
- 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, which is bad)

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: “loop stream detector” (Core 2, Core i7)
- Put entire loop body into a small cache
  - Core2: 18 macro-ops, up to four taken branches
  - Core i7: 28 micro-ops (avoids re-decoding macro-ops!)
- Any branch mis-prediction requires normal re-fetch

Other options: next-next-block prediction, “trace cache”

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 simple and perhaps lower power
  - E.g., TransMeta Crusoe (4-wide)
- Variant: Explicitly Parallel Instruction Computing (EPIC)
  - A bit more flexible encoding & some hardware to help compiler
  - E.g., Intel Itanium (6-wide)

Dynamically-scheduled superscalar (next topic)
- Hardware extracts more ILP by on-the-fly reordering
- Core 2, Core i7 (4-wide), Alpha 21264 (4-wide)
Trends in Superscalar Width

Multiple Issue Recap

Multiple issue
- Exploits insn level parallelism (ILP) beyond pipelining
- Improves IPC, but perhaps at some clock & energy penalty
- 4-6 way issue is about the peak issue width currently justifiable
  - Low-power implementations today typically 2-wide superscalar

Problem spots
- N^2 bypass & register file $\rightarrow$ clustering
- Fetch + branch prediction $\rightarrow$ buffering, loop streaming, trace cache
- N^2 dependency check $\rightarrow$ VLIW/EPIC (but unclear how key this is)

Implementations
- Superscalar vs. VLIW/EPIC