A Key Theme of CIS 371: 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

This Unit: (In-Order) Superscalar Pipelines

- Idea of instruction-level parallelism
- Superscalar hardware issues
  - Bypassing and register file
  - Stall logic
  - Fetch and branch prediction
- “Superscalar” vs VLIW/EPIC

Readings

- P&H
  - Chapter 4.10
“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)

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, *dependencies* 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 src1, src2, -> dest1
  ADD src2, src2, -> dest2 (2 checks)
- For three instructions: 6 checks
  ADD src1, src2, -> dest1
  ADD src2, src2, -> dest2 (2 checks)
  ADD src3, src3, -> dest3 (4 checks)
- For four instructions: 6 checks
  ADD src1, src2, -> dest1
  ADD src2, src2, -> dest2 (2 checks)
  ADD src3, src2, -> dest3 (4 checks)
  ADD src4, src2, -> 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 I₁ is an input register to I₂?**
  - 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
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 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 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

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
    - $X$.op==LOAD $\&\&$ (D.rs1==X.rd $||$ D.rs2==X.rd)
  - Two "terms": $\propto 2N$
- Now: same logic for a 2-wide pipeline
  - $X_1$.op==LOAD $\&\&$ (D.rs1==X_1.rd $||$ D.rs2==X_1.rd) $||$
  - $X_2$.op==LOAD $\&\&$ (D.rs1==X_2.rd $||$ D.rs2==X_2.rd) $||$
  - Eight "terms": $\propto 2N^2$
    - $N^2$ dependence cross-check
    - Not quite done, also need
      - $D_2.rs1==D_1.rd$ $||$ $D_2.rs2==D_1.rd$
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
  - 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 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

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

- 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
  - Most general solution, any two accesses per cycle
    - Lots of wires; expensive in terms of latency, area (cost), and power

- Option\#2: banking (or interleaving)
  - Divide D$ into "banks" (by address), one access per bank per cycle
    - Bank conflict: two accesses to same bank \( \rightarrow \) one stalls
      - Small latency, area, power overheads
      - One access per bank per cycle, assuming no conflicts
      - Complex stall logic \( \rightarrow \) address not known until execute stage
      - To support \( N \) accesses, need \( 2N+ \) banks to avoid frequent conflicts
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
    - Alpha 21264: 4-way superscalar, two clusters

Superscalar “Front End”
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

- Option #3: “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
- Option #4: trace cache (Pentium 4)
  - Tracks “traces” of disjoint but dynamically consecutive instructions
  - Pack (predicted) taken branch & its target into a one “trace” entry
  - Fetch entire “trace” while predicting the “next trace”
Implementations of “Multiple Issue”

Multiple Issue Redux

- 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

- 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

Multiple Issue Summary

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

- Multiple-issue designs
  - "Superscalar" vs VLIW

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