Computer Architecture

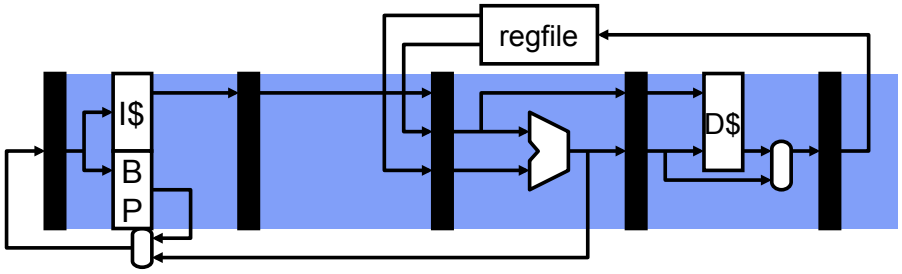
Unit 7: Superscalar Pipelines

Slides developed by Milo Martin & Amir Roth at the University of Pennsylvania
with sources that included University of Wisconsin slides
by Mark Hill, Guri Sohi, Jim Smith, and David Wood

A Key Theme: 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
- Then:
  - 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 & 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, ***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 src1_1, src2_1 -> dest_1
ADD src1_2, src2_2 -> dest_2   (2 checks)
```

- For three instructions: 6 checks

```
ADD src1_1, src2_1 -> dest_1
ADD src1_2, src2_2 -> dest_2   (2 checks)
ADD src1_3, src2_3 -> dest_3   (4 checks)
```

- For four instructions: 12 checks

```
ADD src1_1, src2_1 -> dest_1
ADD src1_2, src2_2 -> dest_2   (2 checks)
ADD src1_3, src2_3 -> dest_3   (4 checks)
ADD src1_4, src2_4 -> dest_4   (6 checks)
```

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

What Checking Is Required?

- For two instructions: 2 checks

```
ADD src1_1, src2_1 -> dest_1
ADD src1_2, src2_2 -> dest_2   (2 checks)
```

- For three instructions: 6 checks

```
ADD src1_1, src2_1 -> dest_1
ADD src1_2, src2_2 -> dest_2   (2 checks)
ADD src1_3, src2_3 -> dest_3   (4 checks)
```

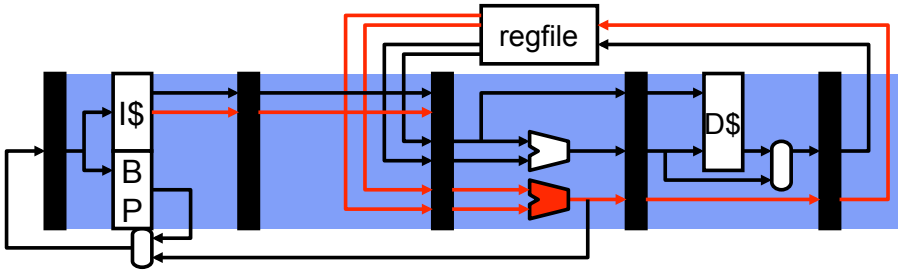
- For four instructions: 12 checks

```
ADD src1_1, src2_1 -> dest_1
ADD src1_2, src2_2 -> dest_2   (2 checks)
ADD src1_3, src2_3 -> dest_3   (4 checks)
ADD src1_4, src2_4 -> dest_4   (6 checks)
```

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

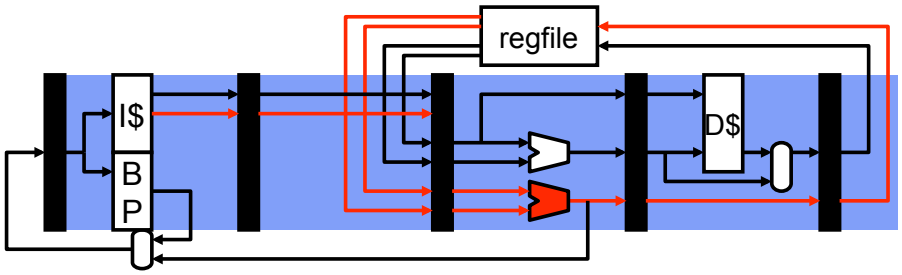
How do we build such "superscalar" hardware?

Multiple-Issue or "Superscalar" Pipeline



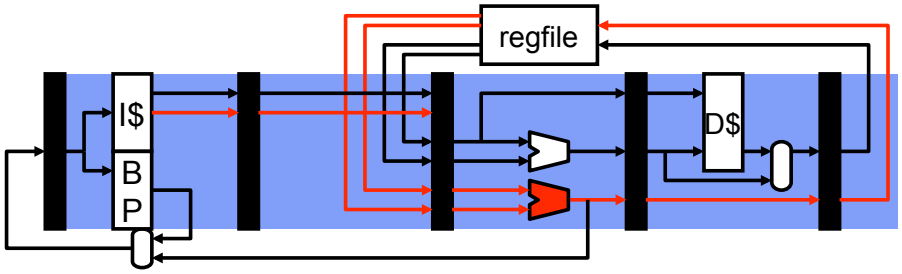
- Overcome this limit using **multiple issue**
  - Also called **superscalar**
  - Two instructions per stage at once, or three, or four, or eight…
  - **"Instruction-Level Parallelism (ILP)"** [Fisher, IEEE TC'81]
- Today, typically "4-wide" (Intel Core i7, AMD Opteron)
  - Some more (Power5 is 5-issue;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

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 Implementation Challenges

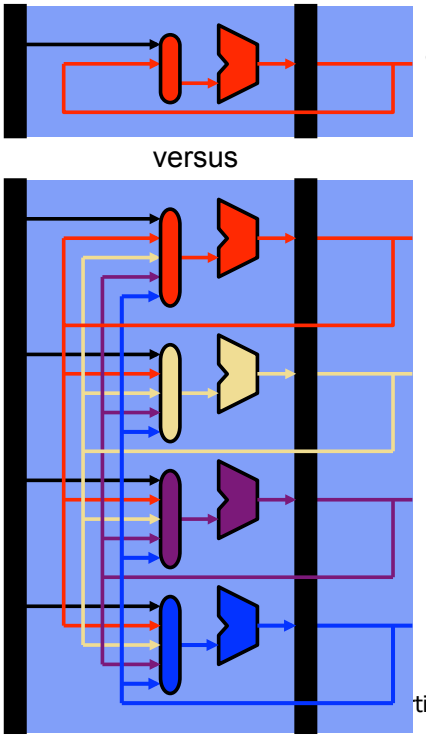
Superscalar Challenges - Front End

- **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 - order $\mathbf{N^2}$ for *N*-wide machine
  - Not all combinations of types of instructions possible
- **Superscalar register read**
  - Port for each register read (4-wide superscalar ➔ 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 insn)
- **Superscalar bypass paths**
  - More possible sources for data values
  - Order (N^2 * 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 ➔ 4 write ports
- **Fundamental challenge:**
  - Amount of ILP (instruction-level parallelism) in the program
  - Compiler must schedule code and extract parallelism

Superscalar Bypass

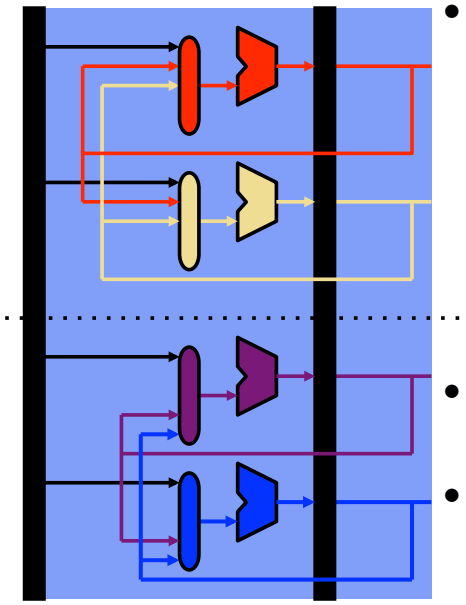


- **N^2 bypass network**
  - N+1 input muxes at each ALU input
  - N^2 point-to-point connections
  - Routing lengthens wires
  - Heavy capacitive load
- And this is just one bypass stage (MX)!
  - There is also WX bypassing
  - Even more for deeper pipelines
- One of the big problems of superscalar
  - Why? On the critical path of single-cycle "bypass & execute" loop

Not All N^2 Created Equal

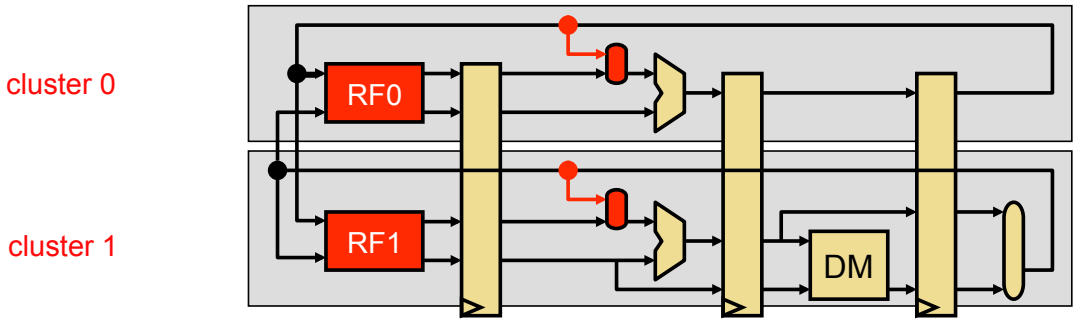
- N^2 bypass vs. N^2 stall logic & dependence cross-check
  - Which is the bigger problem?
- N^2 bypass … by far
  - 64- bit quantities (vs. 5-bit)
  - Multiple levels (MX, WX) of bypass (vs. 1 level of stall logic)
  - Must fit in one clock period with ALU (vs. not)
- Dependence cross-check not even 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 insns to same cluster
- **Cluster register file**, too
  - Replica a register file per cluster
  - All register writes update all replicas
  - Fewer read ports; only for cluster

Mitigating N^2 RegFile: Clustering++

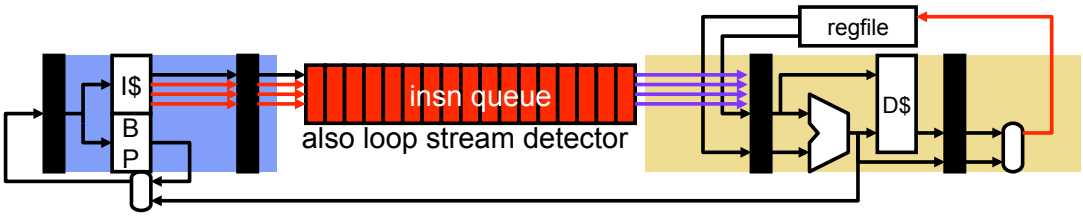


- **Clustering**: split **N**-wide execution pipeline into **K** clusters
  - With centralized register file, 2N read ports and N write ports
- **Clustered register file**: extend clustering to register file
  - Replicate the register file (one replica per cluster)
  - Register file supplies register 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 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
- 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)

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
    - Low-power implementations today typically 2-wide superscalar
- Problem spots
  - N^2 bypass & register file → clustering
  - Fetch + branch prediction → buffering, loop streaming, trace cache
  - N^2 dependency check → VLIW/EPIC (but unclear how key this is)
- Implementations
  - Superscalar vs. VLIW/EPIC

[spacer]