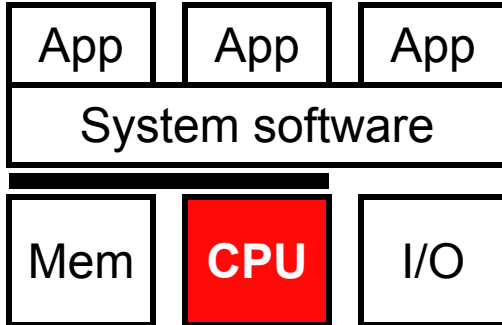


Computer Architecture

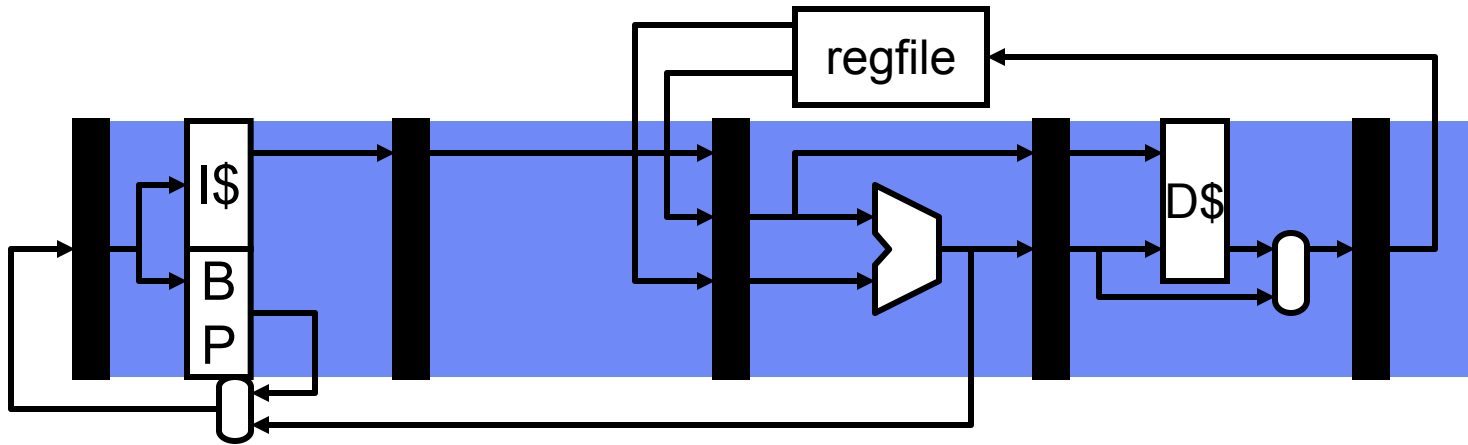
Lec 9: Superscalar Pipelines

(In-Order) Superscalar Pipelines



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

“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 not achievable (due to hazards)
 - Diminishing returns from “super-pipelining” (hazards + overhead)

An Opportunity...

- But consider:

ADD r1, r2 -> r3

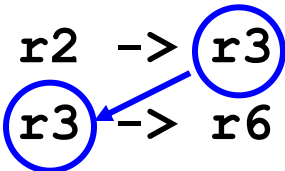
ADD r4, r5 -> r6

- Why not execute them ***at the same time***? (We can!)

- What about:

SUB r1, r2 -> r3

SUB r4, r3 -> r6



- In this case, ***dependences*** prevent parallel execution
- What about three (or more!) 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

What Checking Is Required?

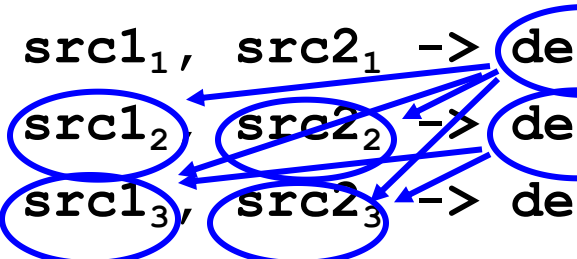
- For two instructions: 2 checks

ADD src1₁, src2₁ -> dest₁
ADD src1₂, src2₂ -> dest₂ (2 checks)



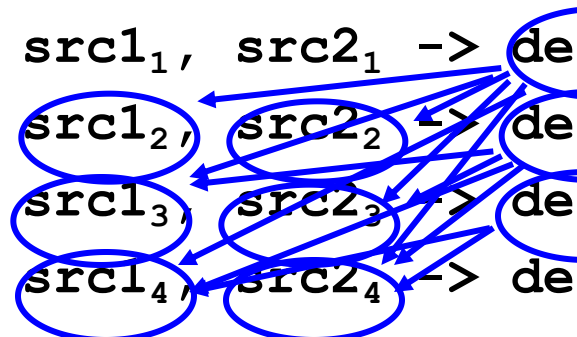
- For three instructions: 6 checks

ADD src1₁, src2₁ -> dest₁
ADD src1₂, src2₂ -> dest₂ (2 checks)
ADD src1₃, src2₃ -> dest₃ (4 checks)



- For four instructions: 12 checks

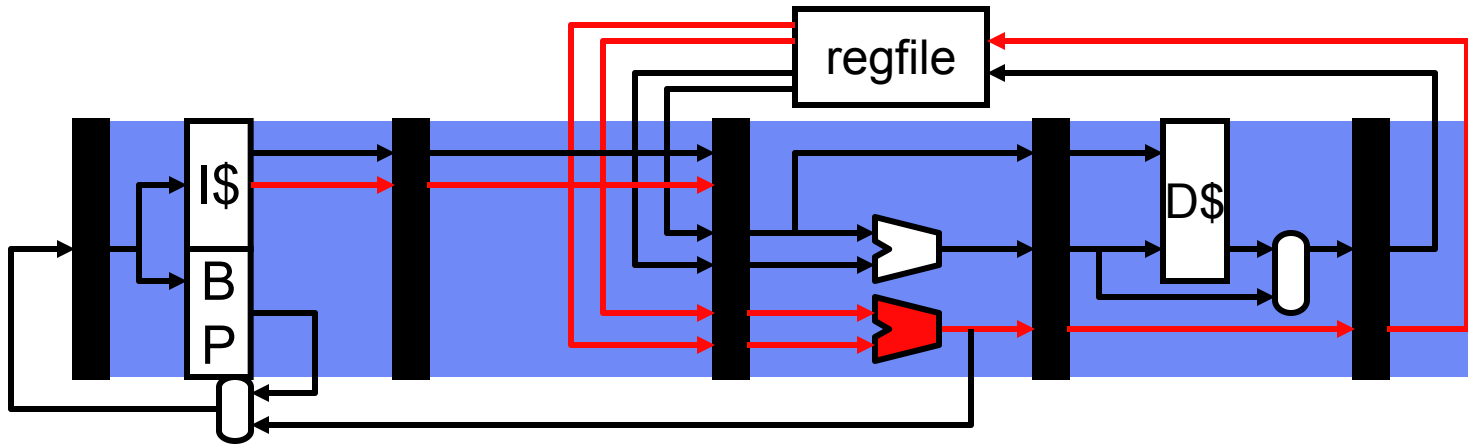
ADD src1₁, src2₁ -> dest₁
ADD src1₂, src2₂ -> dest₂ (2 checks)
ADD src1₃, src2₃ -> dest₃ (4 checks)
ADD src1₄, src2₄ -> dest₄ (6 checks)



- 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-ish-wide” (Intel Broadwell, AMD Ryzen)
 - Broadwell issues up to 8 in the right circumstances, Ryzen up to 6
 - ARM cores usually issue less

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 Pipeline Diagrams - Ideal

scalar

```
lw 0(r1)→r2
lw 4(r1)→r3
lw 8(r1)→r4
add r14,r15→r6
add r12,r13→r7
add r17,r16→r8
lw 0(r18)→r9
```

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1)→r2	F	D	X	M	W							
lw 4(r1)→r3		F	D	X	M	W						
lw 8(r1)→r4			F	D	X	M	W					
add r14,r15→r6				F	D	X	M	W				
add r12,r13→r7					F	D	X	M	W			
add r17,r16→r8						F	D	X	M	W		
lw 0(r18)→r9							F	D	X	M	W	

2-way superscalar

```
lw 0(r1)→r2
lw 4(r1)→r3
lw 8(r1)→r4
add r14,r15→r6
add r12,r13→r7
add r17,r16→r8
lw 0(r18)→r9
```

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1)→r2	F	D	X	M	W							
lw 4(r1)→r3	F	D	X	M	W							
lw 8(r1)→r4		F	D	X	M	W						
add r14,r15→r6		F	D	X	M	W						
add r12,r13→r7			F	D	X	M	W					
add r17,r16→r8			F	D	X	M	W					
lw 0(r18)→r9				F	D	X	M	W				

Superscalar Pipeline Diagrams - Realistic

scalar

```
lw 0(r1) → r2
lw 4(r1) → r3
lw 8(r1) → r4
add r4, r5 → r6
add r2, r3 → r7
add r7, r6 → r8
lw 4(r8) → r9
```

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1) → r2	F	D	X	M	W							
lw 4(r1) → r3		F	D	X	M	W						
lw 8(r1) → r4			F	D	X	M	W					
add r4, r5 → r6				F	D	d*	X	M	W			
add r2, r3 → r7					F	d*	D	X	M	W		
add r7, r6 → r8							F	D	X	M	W	
lw 4(r8) → r9								F	D	X	M	W

2-way superscalar

```
lw 0(r1) → r2
lw 4(r1) → r3
lw 8(r1) → r4
add r4, r5 → r6
add r2, r3 → r7
add r7, r6 → r8
lw 4(r8) → r9
```

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1) → r2	F	D	X	M	W							
lw 4(r1) → r3	F	D	X	M	W							
lw 8(r1) → r4		F	D	X	M	W						
add r4, r5 → r6		F	D	d*	d*	X	M	W				
add r2, r3 → r7			F	D	d*	X	M	W				
add r7, r6 → r8			F	d*	d*	D	X	M	W			
lw 4(r8) → r9				F	d*	D	d*	X	M	W		

Superscalar Implementation Challenges

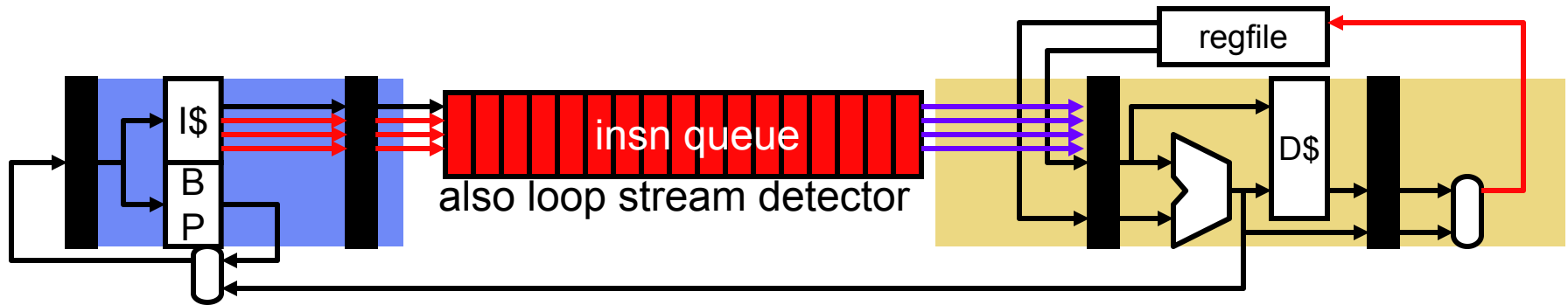
Superscalar Challenges - Front End

- **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 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 $\propto \#ports^2$

Challenges of 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 $\rightarrow \sim 10$ 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

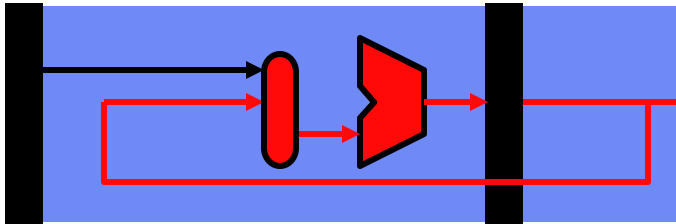


- Option #1: over-fetch and buffer
 - Add a queue between fetch and decode (18 entries in Intel Core2)
 - Compensates for cycles that fetch less than maximum instructions
 - “decouples” the “front end” (fetch) from the “back end” (execute)
- Option #2: “loop stream detector” (Core 2, Core i7)
 - Put 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”

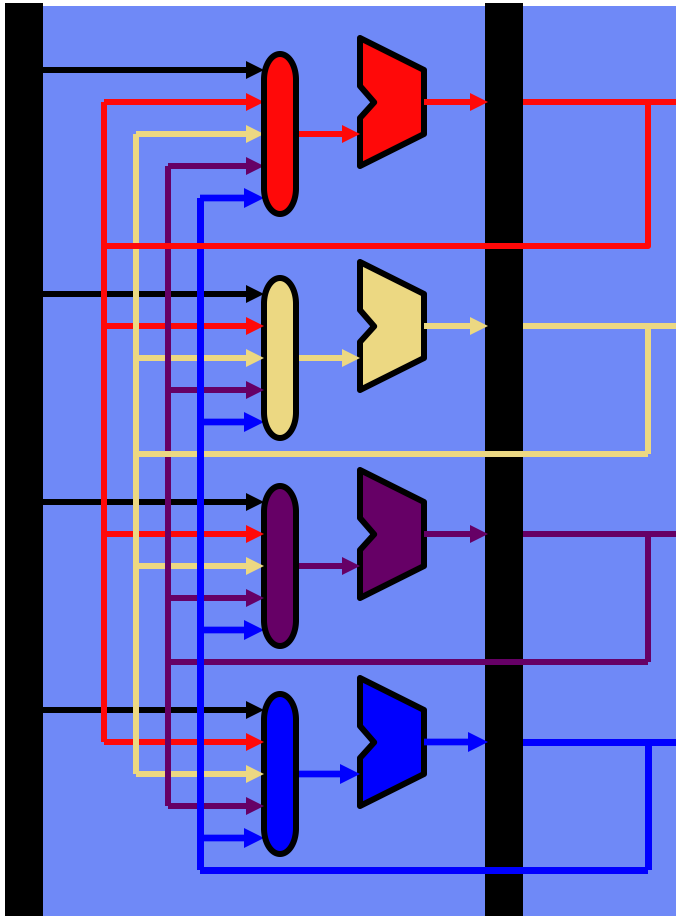
Superscalar Challenges - Back End

- **Superscalar instruction execution**
 - Replicate arithmetic units (but not all, for example, integer divider)
 - Perhaps multiple cache ports (slower access, higher energy)
 - Only for 4-wide or larger (why? only $\sim 35\%$ 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 \rightarrow 4 write ports
- **Fundamental challenge:**
 - Amount of ILP (instruction-level parallelism) in the program
 - Compiler must schedule code and extract parallelism

Superscalar Bypass



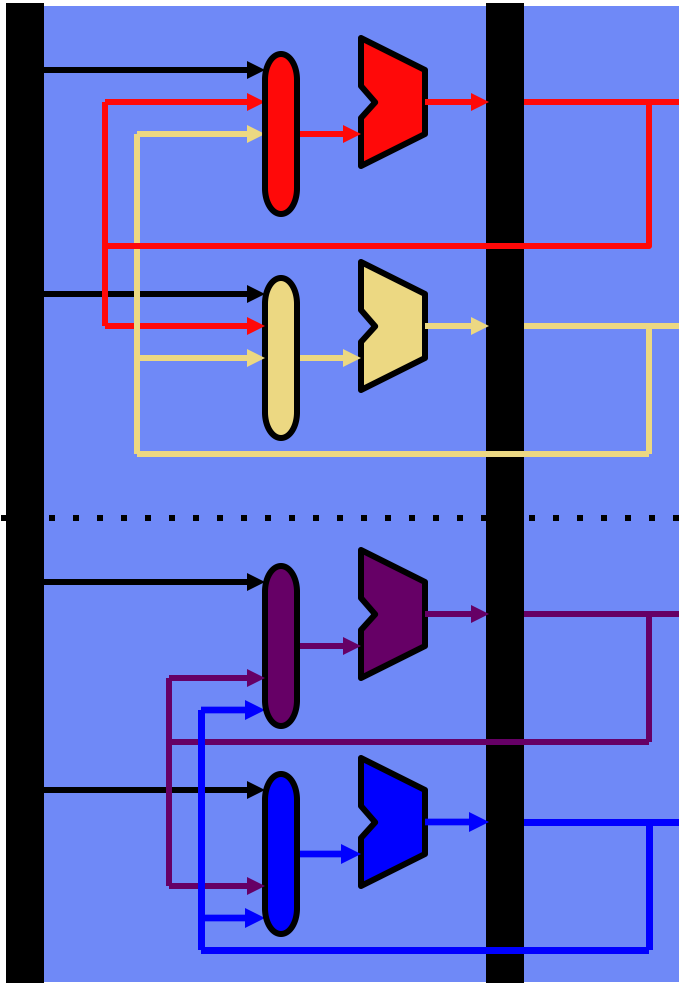
versus



- **N^2 bypass network**

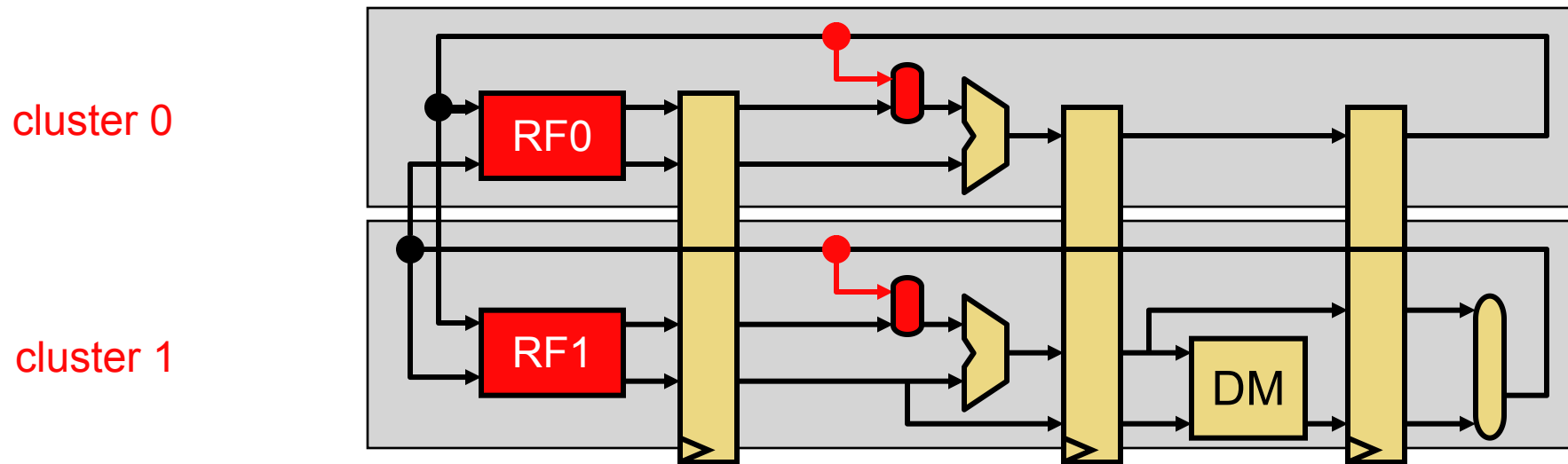
- $N+1$ input muxes at each ALU input
 - N^2 point-to-point connections
 - Routing lengthens wires
 - Heavy capacitive load
- And this is just one bypass stage (MX)!
 - There is also WX bypassing
 - Even more for deeper pipelines
 - One of the big problems of superscalar
 - Why? On the critical path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

Mitigating N² Bypass & Register File



- **Clustering**: mitigates N² 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
 - Replicate a register file per cluster
 - All register writes update all replicas
 - Fewer read ports; only for cluster

Mitigating N^2 RegFile with Clustering

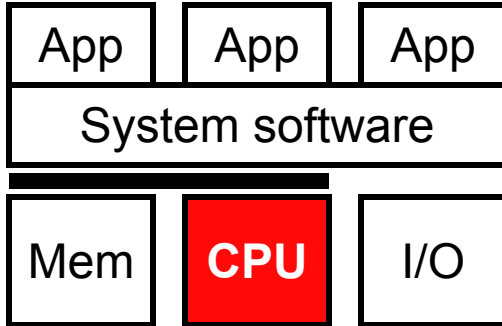


- **Clustering**: split N -wide execution pipeline into K clusters
 - With centralized register file, $2N$ read ports and N write ports
- **Clustered register file**: extend clustering to register file
 - Replicate the register file (one replica per cluster)
 - Register file supplies register operands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

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), most DSPs
 - **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**
 - Intel Atom/Core/Xeon, AMD Opteron/Ryzen, some ARM A-series

(In-Order) Superscalar Pipelines



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