CSE 548: Computer Systems Architecture

Superscalar Processors

Luis Ceze, Spring 2017

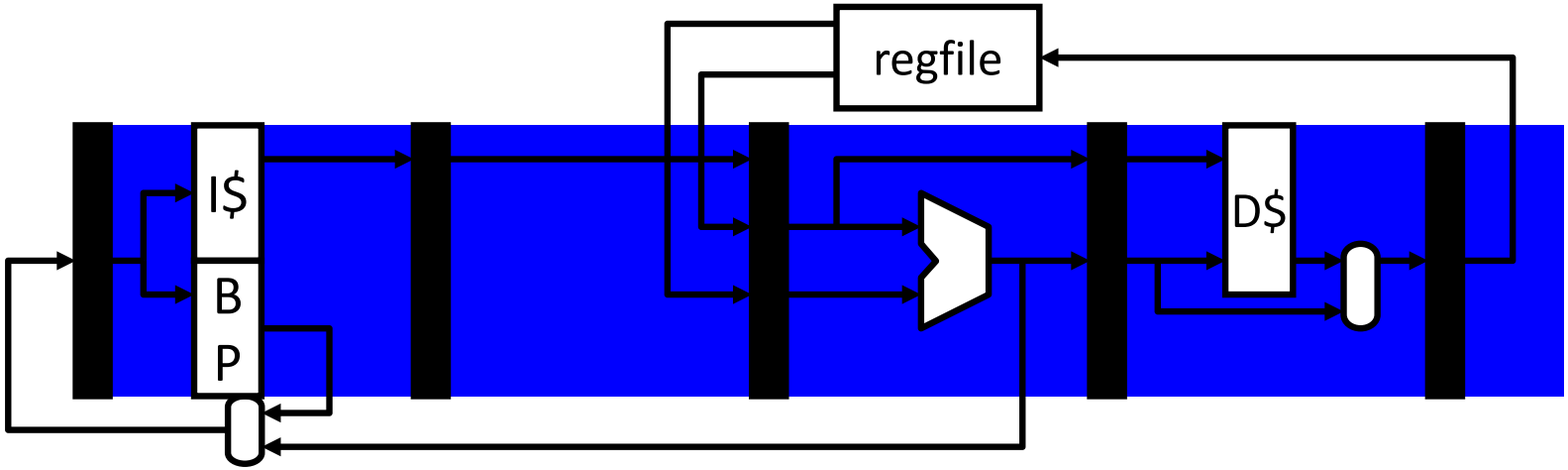
(based on slides lifted from friends at UPenn, UIUC, UW, MIT)

Announcements

This Unit: Superscalar Execution

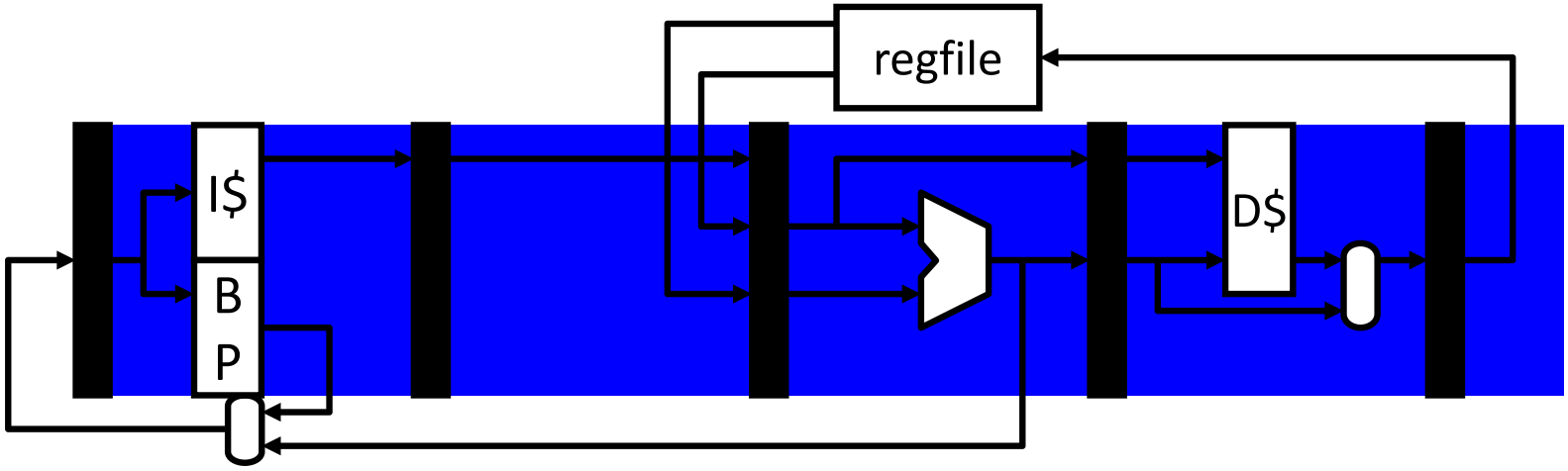
- Superscalar scaling issues
  - Multiple fetch and branch prediction
  - Dependence-checks & stall logic
  - Wide bypassing
  - Register file & cache bandwidth
- Multiple-issue designs
  - “Superscalar”
  - VLIW and EPIC (Itanium)

How to make our pipelined design faster?



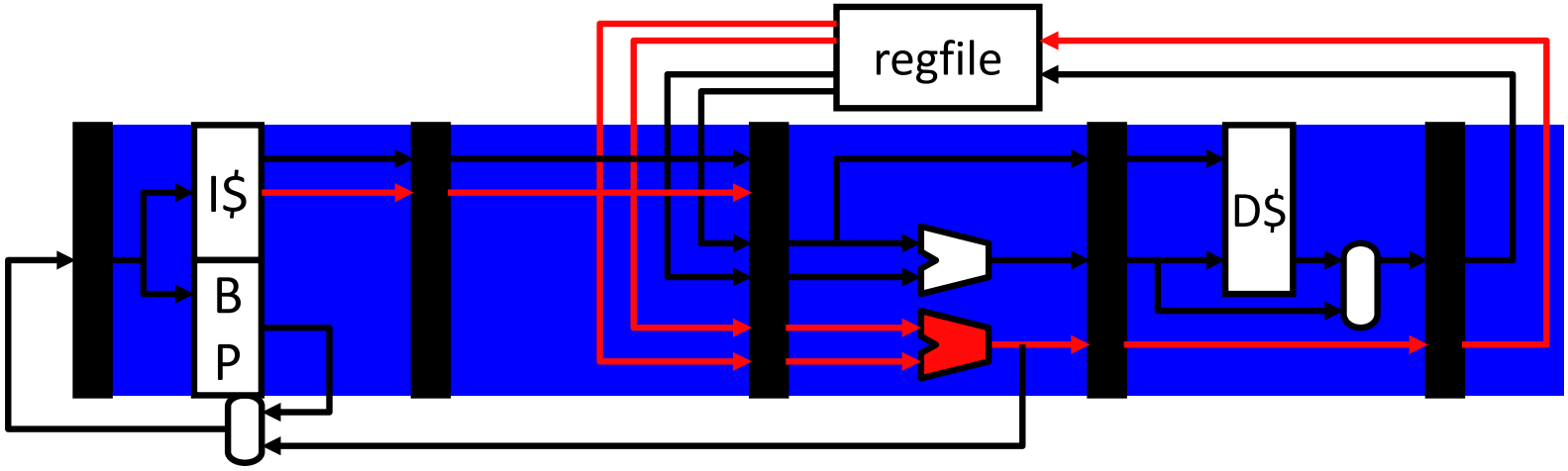
- What happens if we make the pipeline deeper? (more stages)

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)

Superscalar Pipeline Diagrams - Ideal

scalar

	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

	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

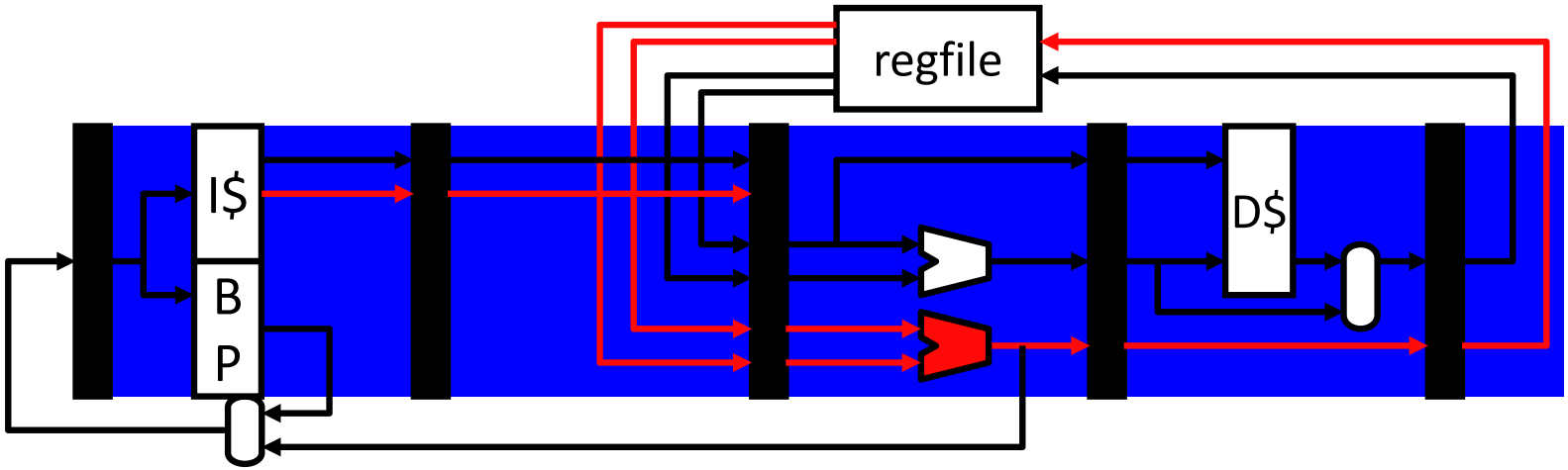
scalar

	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	X	M	W		
`add r7,r6➔r8`							F	D	X	M	W	
`lw 0(r8)➔r9`								F	D	X	M	W

2-way superscalar

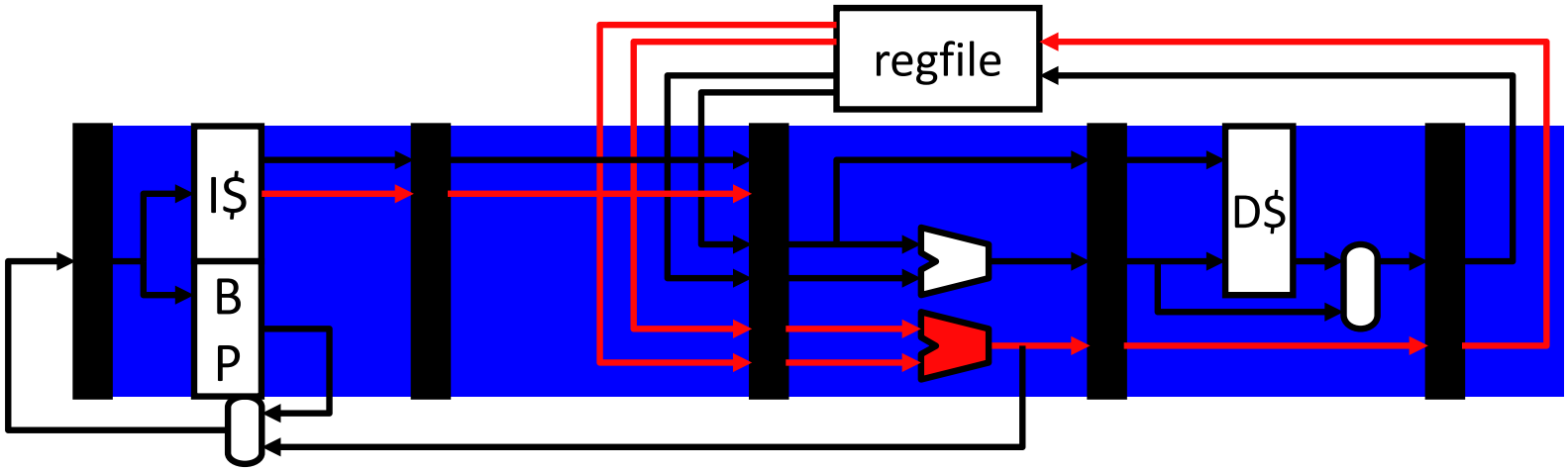
	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)➔r2`	F	D	X	M	W							
`lw 4(r1)➔r3`	F	D	X	M	W							
`lw 8(r1)➔r4`		F	D	X	M	W						
`add r4,r5➔r6`		F	d*	d*	D	X	M	W				
`add r2,r3➔r7`			F	d*	D	X	M	W				
`add r7,r6➔r8`					F	D	X	M	W			
`lw 0(r8)➔r9`					F	d*	D	X	M	W		

A Typical Dual-Issue Pipeline



- Fetch an entire 16B or 32B cache block
  - 4 to 8 instructions (assuming 4-byte fixed length instructions)
  - 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)
- *What are the added costs of this design?*

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 Challenges - Front End

- **Wide instruction fetch**
  - Modest: need multiple instructions per cycle
  - Aggressive: predict multiple branches, trace cache
- **Wide instruction decode**
  - Replicate decoders
- **Wide instruction issue** (submit for execution)
  - Determine when instructions can proceed in parallel
  - Not all combinations possible
  - More complex stall logic - order N^2 for *N*-wide machine
- **Wide register read**
  - One port for each register read
    - Each port needs its own set of address and data wires
  - Example, 4-wide superscalar ➔ 8 read ports

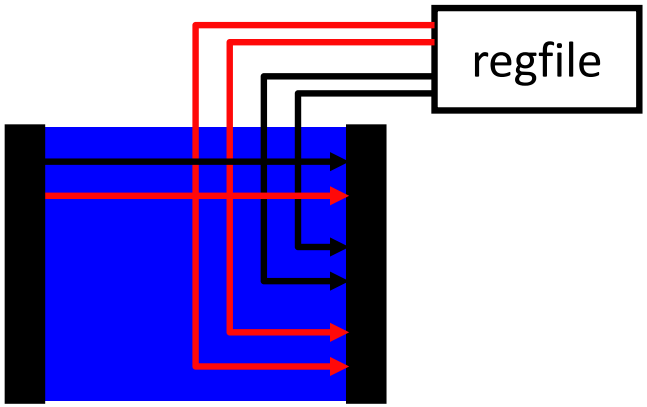
Superscalar Challenges - Back End

- **Wide instruction execution**
  - Replicate arithmetic units
  - Multiple cache ports
- **Wide instruction register writeback**
  - One write port per instruction that writes a register
  - Example, 4-wide superscalar ➔ 4 write ports
- **Wide bypass paths**
  - More possible sources for data values
  - Order ($N^2 * P$) for *N*-wide machine with execute pipeline depth *P*

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

Superscalar Execution

Superscalar Decode and 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?
  - 2N register read ports (latency $\propto$ #ports)
  - \+ Actually less than 2N, most values come from bypasses
  - More about this in a bit
- What about the **stall logic**? (i.o.w., what happens to dependence check?)

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"
- Now: same logic for a 2-wide pipeline

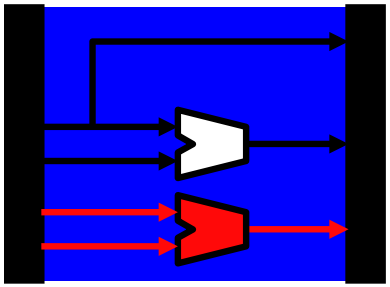
  X/M_1.op==LOAD && (D/X_1.rs1==X/M_1.rd || D/X_1.rs2==X/M_1.rd) ||

  X/M_1.op==LOAD && (D/X_2.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) ||

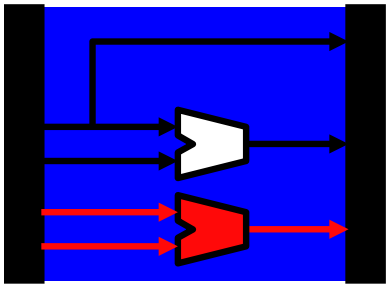
  X/M_2.op==LOAD && (D/X_2.rs1==X/M_2.rd || D/X_2.rs2==X/M_2.rd)
  - Eight "terms": $\propto 2N^2$
    - **N^2 dependence cross-check**
  - Not quite done, also need
    - D/X_2.rs1==D/X_1.rd || D/X_2.rs2==D/X_1.rd

Superscalar Execute



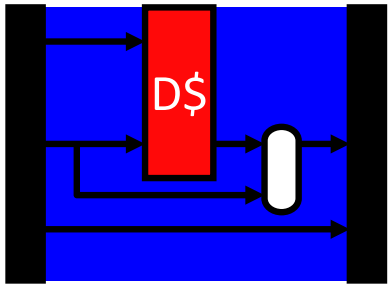
- What is involved in executing N insns per cycle?

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 FP dividers? No, FP dividers are huge and **`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 ALU
    - Alpha 21164: 2 integer (including 2 loads) + 2 FP

Superscalar Memory Access



- What about multiple loads/stores per cycle?
  - Probably only necessary on processors 4-wide or wider– ***why?***
  - More important to support multiple loads than multiple stores
    - Insn mix: loads (~20–25%), stores (~10–15%)

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?

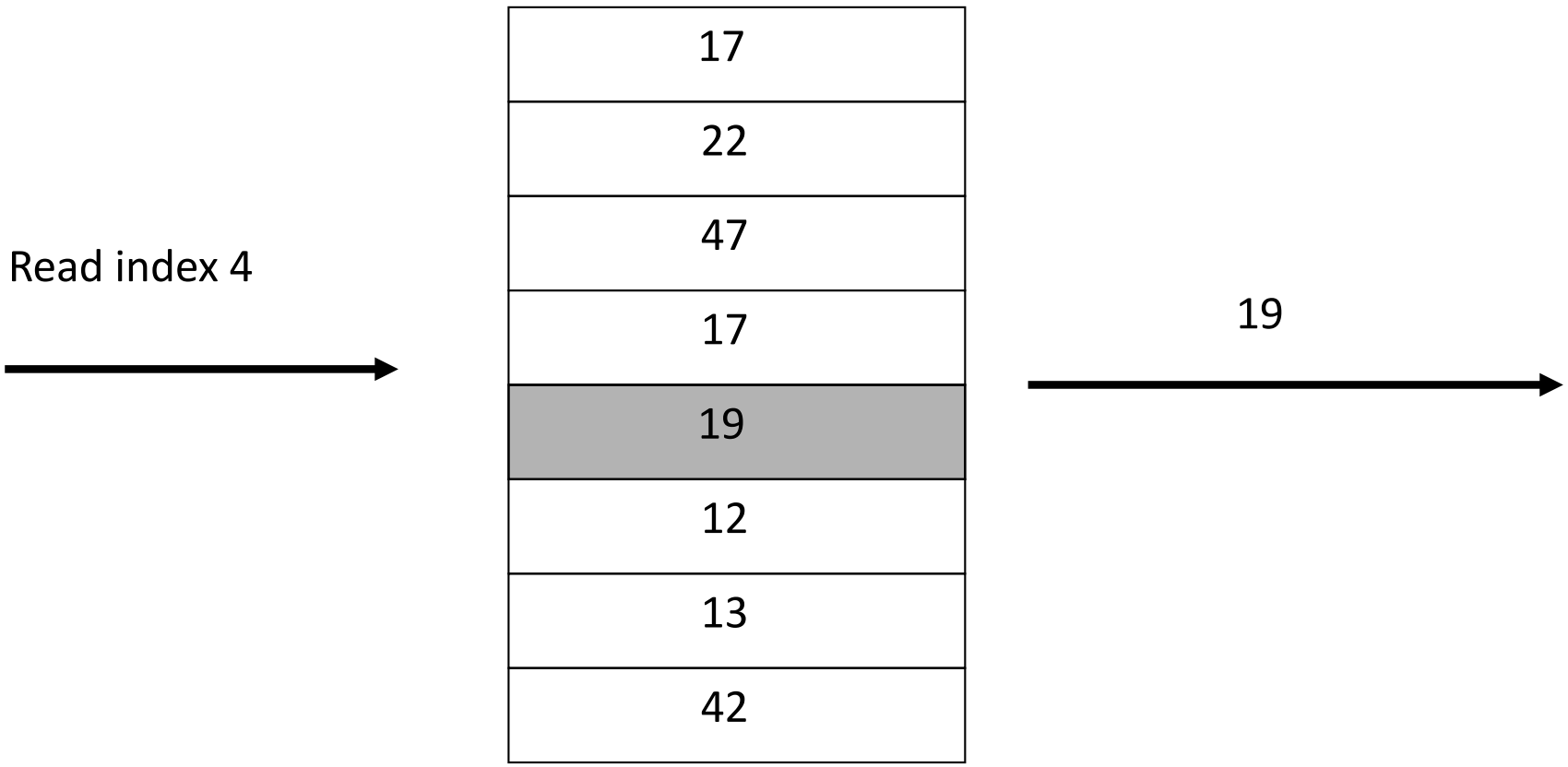
- Option#1: **multi-porting**
  - \+ Most general solution, any two accesses per cycle
  - – Expensive in terms of latency, area (cost), and power

- Option #2: **replication**
  - – Additional read bandwidth only, but writes must go to all replicas
  - \+ General solution for loads, no latency penalty
  - – Not a solution for stores (that's OK), area (cost), power penalty
    - Is this what Alpha 21164 does?

RAM vs CAM

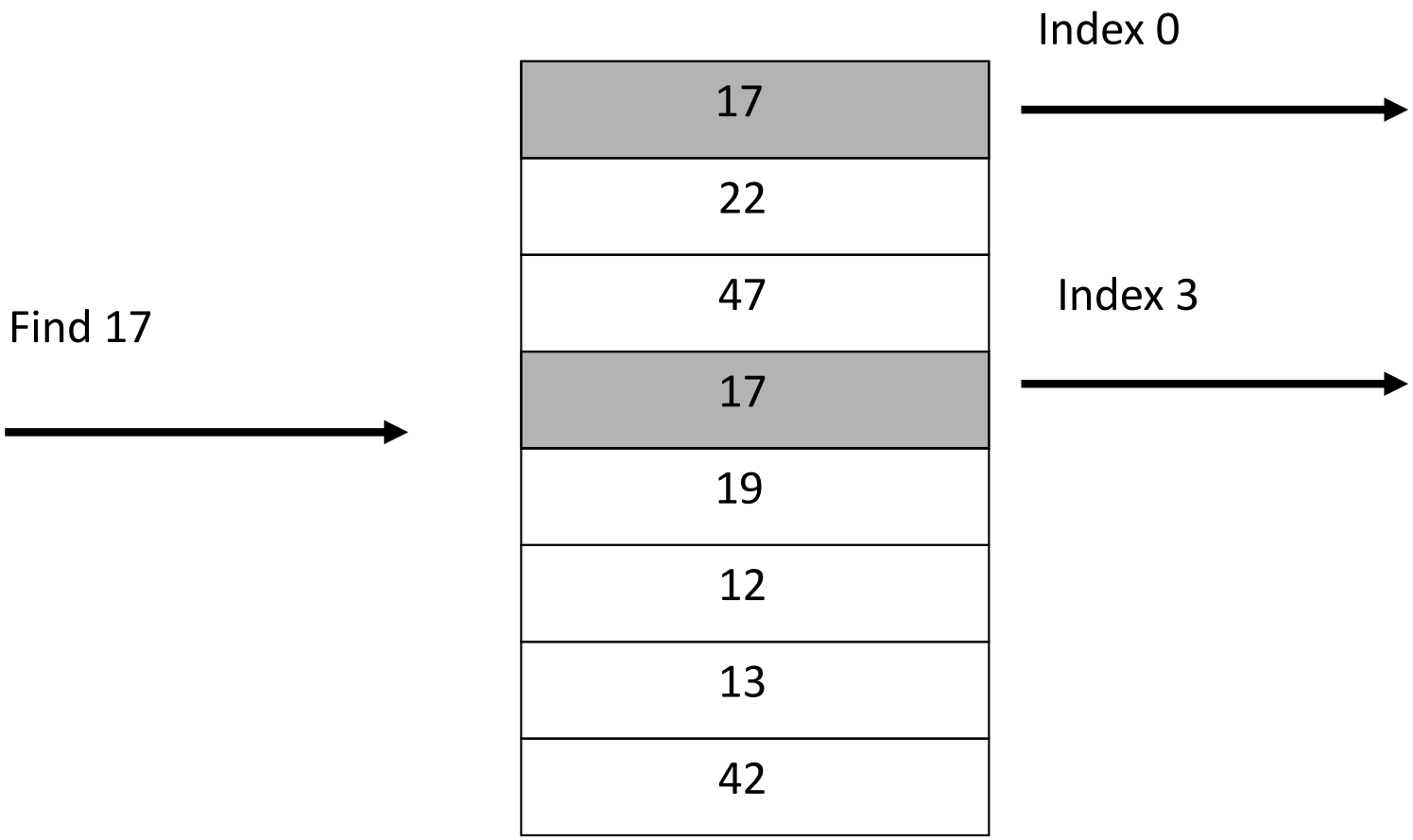
- Random Access Memory
  - Read/write specific index
  - Get/set value there
- Content Addressable Memory
  - Search for a value
  - Find matching indices
- One structure can have ports of both types

RAM vs CAM: RAM



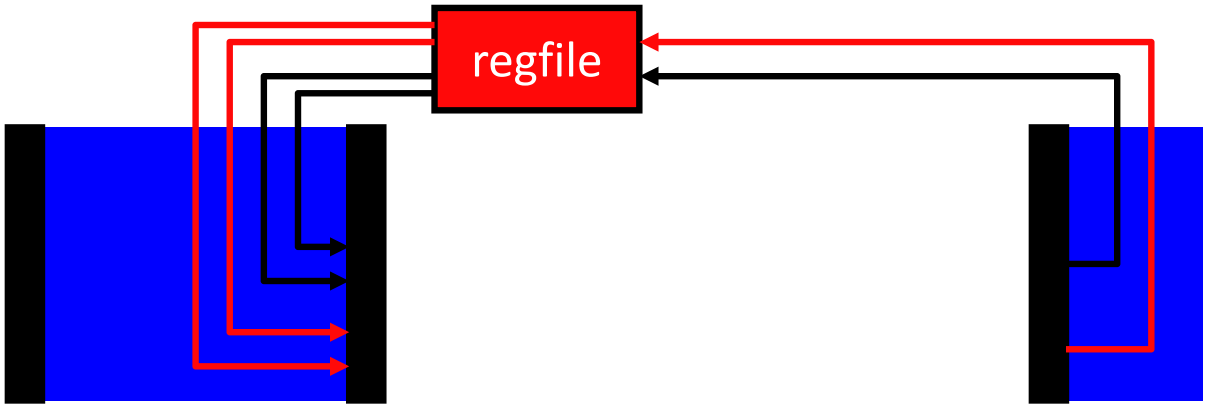
RAM: read/write specific index

RAM vs CAM: CAM



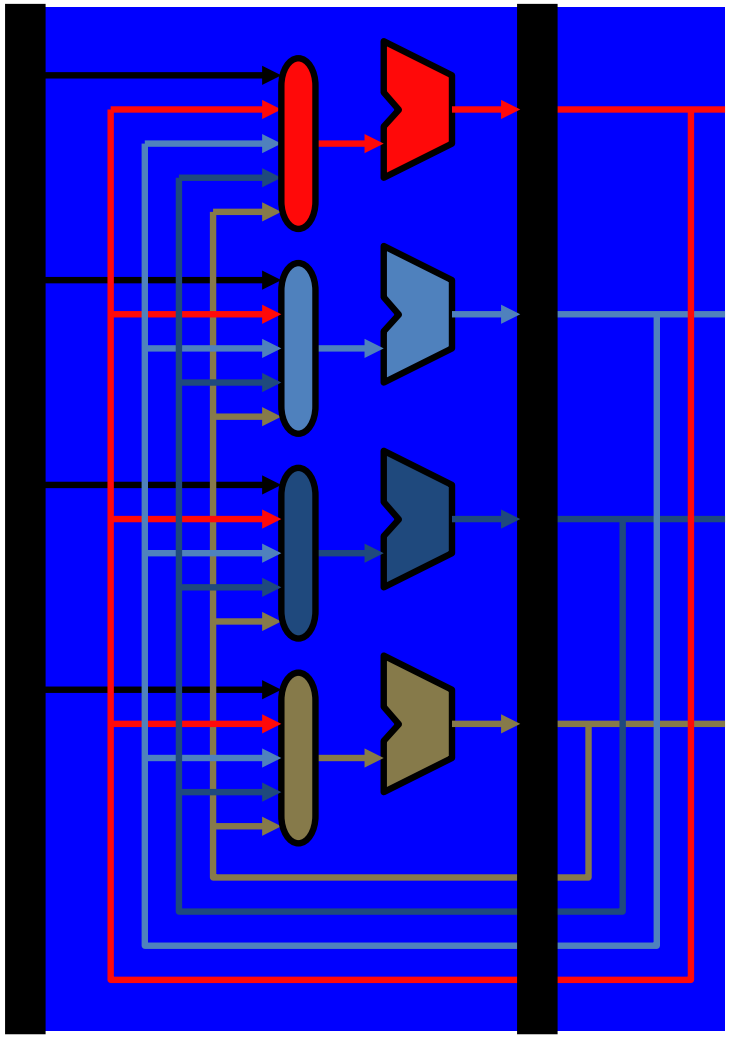
CAM: search for value

Superscalar Register Read/Write



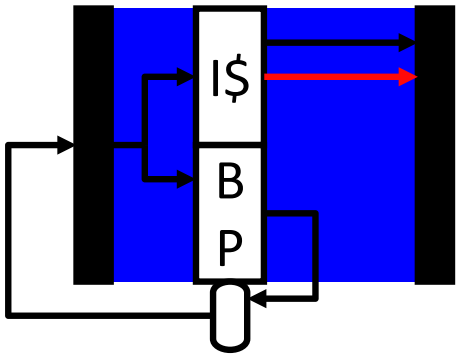
- How many register file ports to execute N insns per cycle?
  - Nominally, 2N read + N write (2 read + 1 write per insn)
    - Latency, area $\propto$ #ports^2
  - In reality, fewer than that
    - Read ports: many values come from bypass network
    - Write ports: stores, branches (35% insns) don't write registers
- Replication works great for regfiles (used in Alpha 21164)
- Banking? Not so much

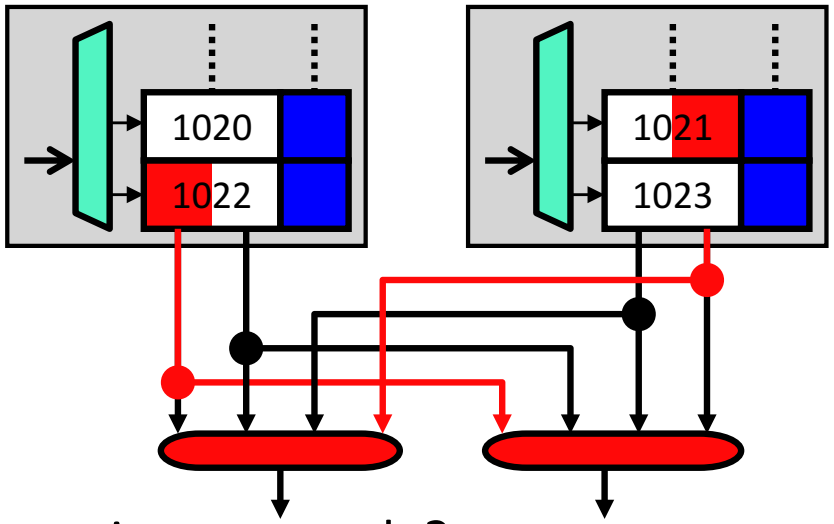
Superscalar Bypass



- **N^2 bypass network**
  - N+1 input muxes at each ALU input
  - N^2 point-to-point connections
  - Routing lengthens wires
  - Expensive metal layer crossings
  - 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

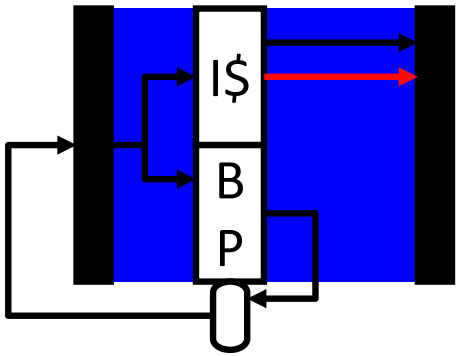
Superscalar Fetch

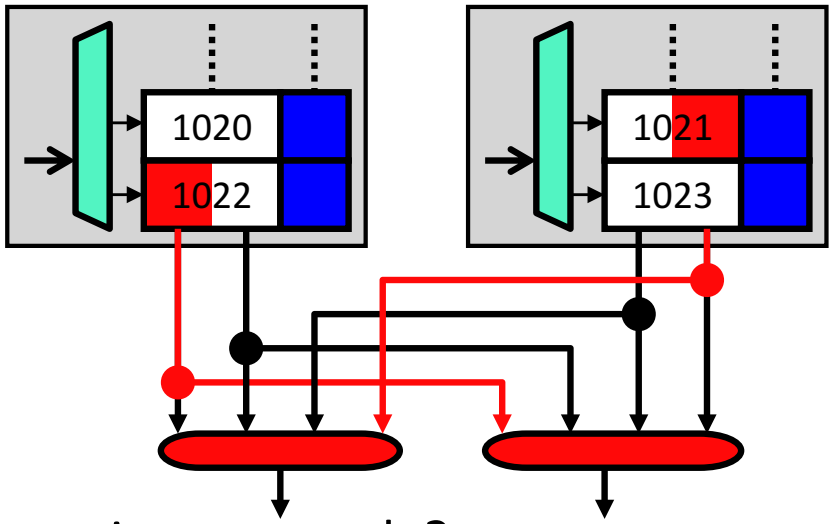




- What is involved in fetching multiple instructions per cycle?
- In same cache block? → no problem
  - Favors larger block size (independent of hit rate)
- Can compilers help? How?
- In multiple blocks? Hmm, how?

Superscalar Fetch





- What is involved in fetching multiple instructions per cycle?
- In same cache block? → no problem
  - Favors larger block size (independent of hit rate)
- Compilers align basic blocks to I$ lines (`.align` assembly directive)
  - Reduces I$ capacity
  + Increases fetch bandwidth utilization (more important)
- In multiple blocks? → Fetch block A and A+1 in parallel
  - Banked I$ + **combining network**
  - May add latency (add pipeline stages to avoid slowing down clock)

Wide Non-Sequential Fetch

- Two related questions
  - How many branches predicted per cycle?
  - Can we fetch across the branch if it is predicted “taken”?

Wide Non-Sequential Fetch

- Two related questions
  - How many branches predicted per cycle?
  - Can we fetch across the branch if it is predicted "taken"?
- Simplest, most common organization: "1" and "No"
  - One prediction, discard post-branch insns if prediction is "taken"
  - Lowers effective fetch width and IPC
  - Average number of instructions per taken branch?
    - Assume: 20% branches, 50% taken $\rightarrow$ ~10 instructions
  - Consider a 10-instruction loop body with an 8-issue processor
    - Without smarter fetch, ILP is limited to 5 (not 8)
- Compiler can help
  - Reduce taken branch frequency (e.g., unroll loops)

Branch Prediction and Wide Execution

- What happens to the cost of a branch misprediction in a superscalar processor?

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*0.75*2 = 1.3 → 1.3/1 = 1.3 → 30% slowdown
- 2-way superscalar pipeline
  - **0.5** + 0.2*0.75*2 = 0.8 → 0.8/0.5 = 1.6 → 60% slowdown
- 4-way superscalar
  - **0.25** + 0.2*0.75*2 = 0.55 → 0.55/0.25 = 2.2 → 120% slowdown

Multiple-Issue Implementations

- **Statically-scheduled (in-order) superscalar**
  - \+ 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)**
  - \+ Hardware can be dumb and low power
  - – Compiler must group parallel insns, requires new binaries
  - – E.g., TransMeta Crusoe (4-wide)
- **Explicitly Parallel Instruction Computing (EPIC)**
  - – A compromise: compiler does some, hardware does the rest
  - – E.g., Intel Itanium (6-wide)
- **Dynamically-scheduled superscalar**
  - – Pentium Pro/II/III (3-wide), Alpha 21264 (4-wide)

VLIW

- Hardware-centric multiple issue problems
  - Wide fetch+branch prediction, N^2 bypass, N^2 dependence checks
  - Hardware solutions have been proposed: clustering, trace cache

- Software-centric: **very long insn word (VLIW)**
  - Effectively, a 1-wide pipeline, but unit is an N-insn group
  - Compiler guarantees insns within a VLIW group are independent
    - If no independent insns, slots filled with **`nops`**
  - Group travels down pipeline as a unit
    + Simplifies pipeline control (no rigid vs. fluid business)
    + Cross-checks within a group un-necessary
    - Downstream cross-checks still necessary
  - Typically "slotted": 1st insn must be ALU, 2nd mem, etc.
    + Further simplification

What Does VLIW Actually Buy You?

+ Simpler I$/branch prediction
+ Slightly simpler dependence check logic
- Doesn't help bypasses or regfile
  - Which are the much bigger problems
  - Although clustering and replication can help VLIW, too
- Not compatible across machines of different widths
  - Is non-compatibility worth all of this?

- How did TransMeta deal with compatibility problem?
  - Dynamically translates x86 to internal VLIW

Trends in Single-Processor Multiple Issue

	486	Pentium	PentiumII	Pentium4	Itanium	Itanium II	Core2	Core i7 (Sandy B)	Core M (Broadwell)
Year	1989	1993	1998	2001	2002	2004	2006	2011	2015
Width	1	2	3	3	3	6	4	F6/4	4

- Issue width has saturated at 4-6 for high-performance cores
  - Canceled Alpha 21464 was 8-way issue
  - No justification for going wider
  - Out-of-order execution (or EPIC) needed to exploit 4-6 effectively
- For high-performance/power cores, issue width is ~2
  - Out-of-order execution not needed
  - Multi-threading (a little later) helps cope with cache misses