
CSE 530

Fundamentals of Computer Architecture

Spring 2021

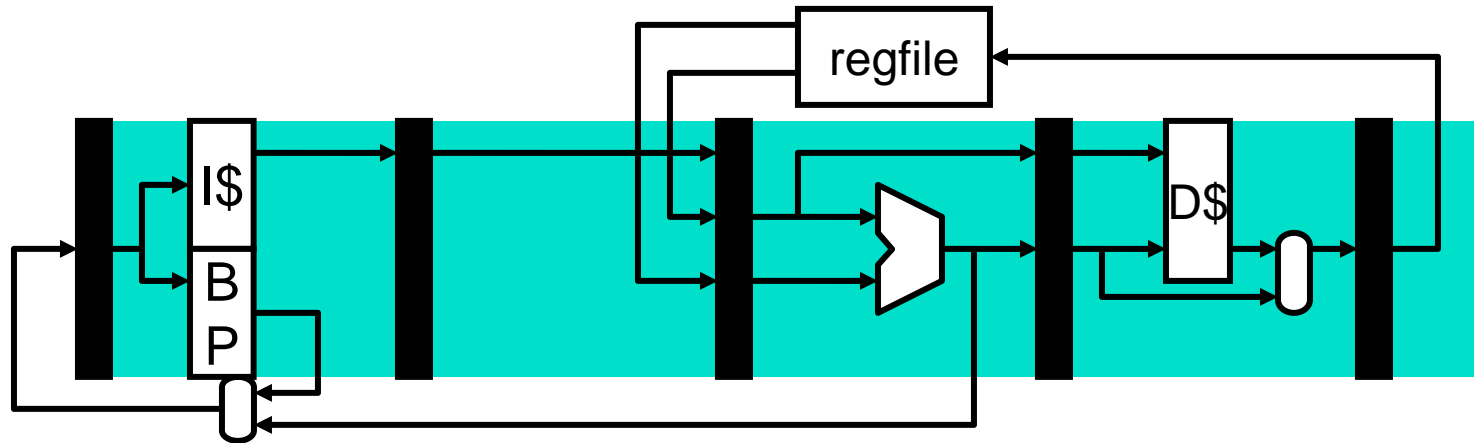
SuperScalar In-Order Datapaths

John (Jack) Sampson (cse.psu.edu/~sampson)

Course material on Canvas

[Adapted in part from Mary Jane Irwin, V. Narayanan, Amir Roth, Milo Martin,
and others]

Scalar pipelines and the Flynn bottleneck



- ❑ So far we have looked at **scalar pipelines**
 - ❑ Fetch and **issue** (send to decode) **one** instruction at a time, so only have one instruction occupying a pipeline stage
 - With bypassing, control speculation (branch prediction), etc.
 - Performance limit (aka “Flynn Bottleneck”) is $CPI = IPC = 1$
 - Limit is never even achieved (hazards)
 - Diminishing returns from “super-pipelining” (hazards + overhead)

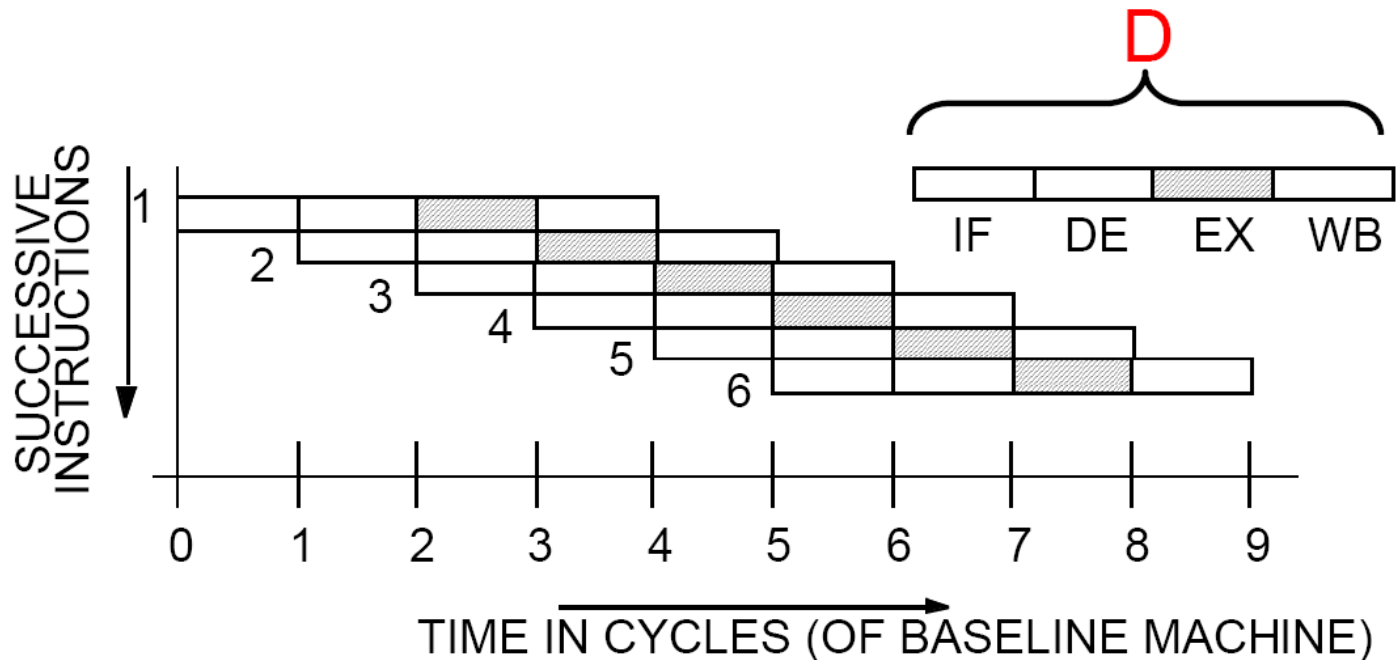
Architectures and Nomenclature for Wide(r)-Issue

Scalar Pipeline (baseline)

Instruction Parallelism = **D**

Operation Latency = **1**

Peak IPC = **1**



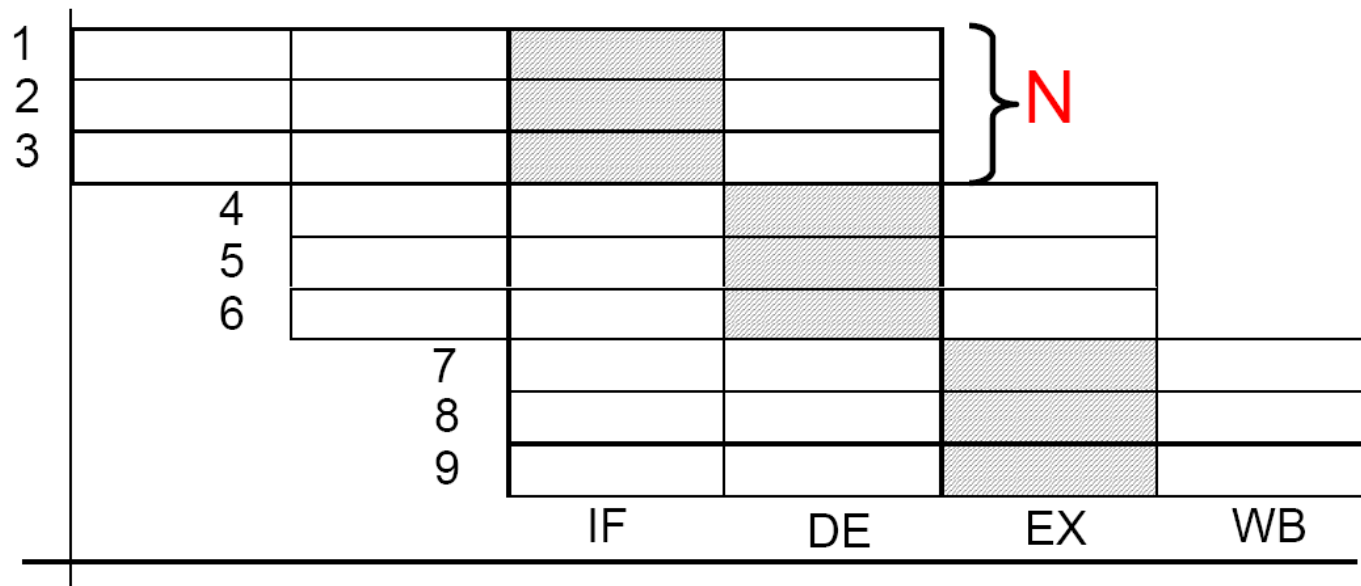
Superscalar Machine

Superscalar (Pipelined) Execution

IP = $D \times N$

OL = *1 baseline cycles*

Peak IPC = *N per baseline cycle*

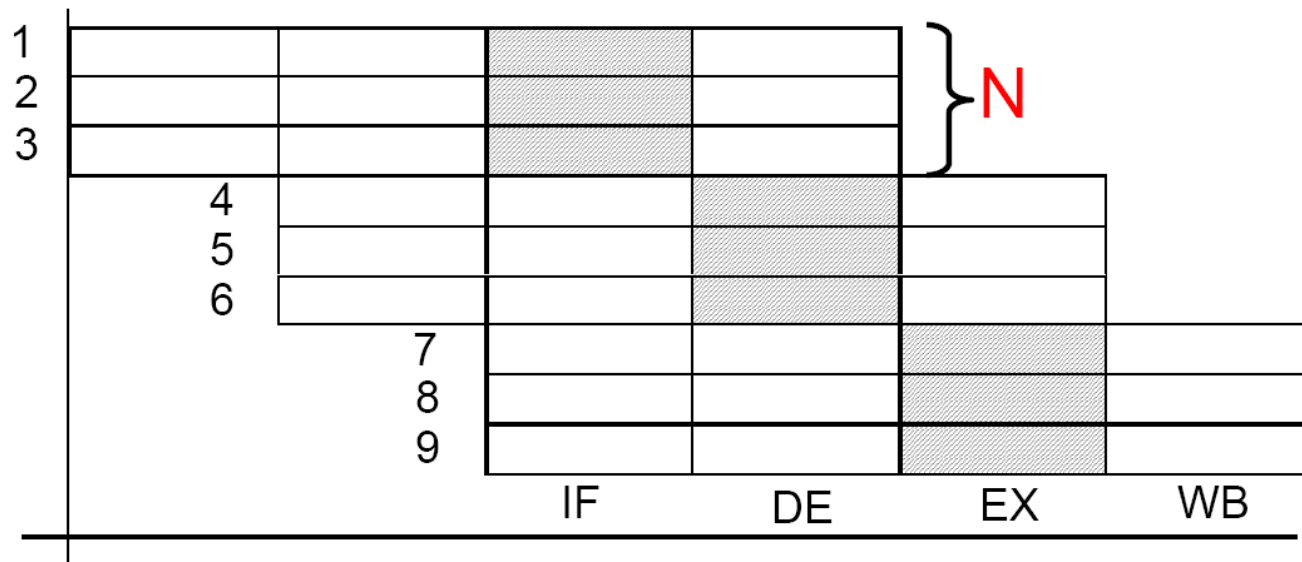


How far can this take us?

CPI of in-order pipelines degrades very sharply if the machine parallelism is increased beyond a certain point, *i.e., when $N \times M$ approaches average distance between dependent instructions*

Forwarding is no longer effective

Pipeline may never be full due to frequent dependency stalls!



ILP: Instruction-Level Parallelism

ILP is a measure of the amount of inter-dependencies between instructions

Average ILP = $\text{no. instruction} / \text{no. cyc required}$

code1: ILP = 1

i.e. must execute serially

code2: ILP = 3

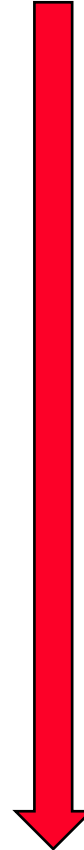
i.e. can execute at the same time

code1: $r1 \leftarrow r2 + 1$
 $r3 \leftarrow r1 / 17$
 $r4 \leftarrow r0 - r3$

code2: $r1 \leftarrow r2 + 1$
 $r3 \leftarrow r9 / 17$
 $r4 \leftarrow r0 - r10$

Purported Limits on ILP

Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86
Tjaden and Flynn [1973]	1.96
Uht [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin and Patt [1991]	6
Wall [1991]	7
Kuck et al. [1972]	8
Riseman and Foster [1972]	51
Nicolau and Fisher [1984]	90



Large range 1.6 – 90!

Reason:

Equally large range of assumptions!

ILP limits are fundamentally program + compilation dependent independent of the ability of hardware to exploit ILP

Multiple-issue implementations

- ❑ Fetch and issue **more than one** instruction in a cycle

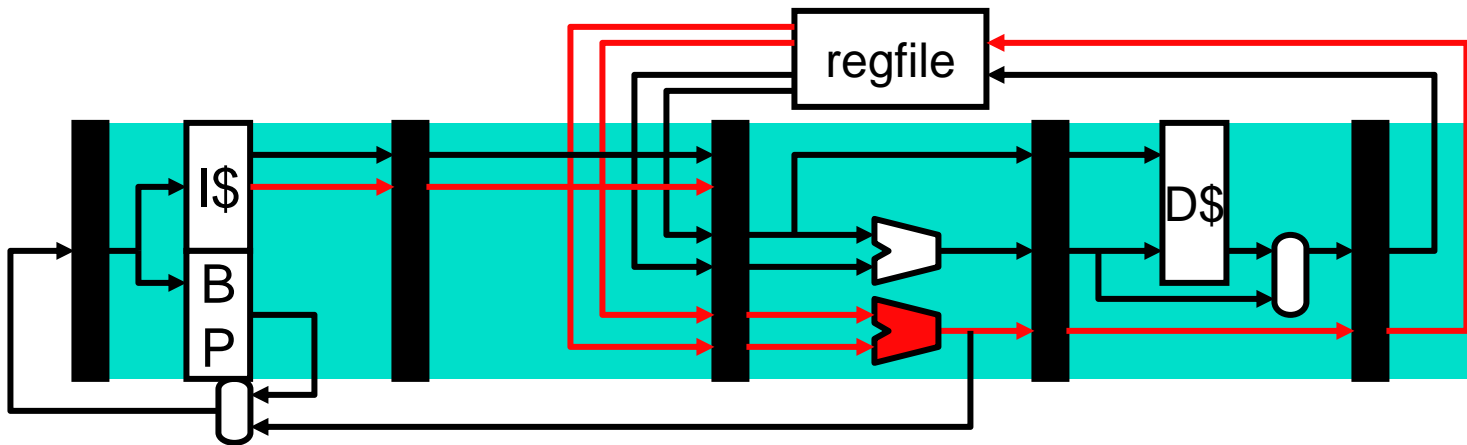
1. **Statically-scheduled (in-order) SuperScalar**

- Hardware must figure out what can be done in parallel
- + Executes unmodified sequential programs
- E.g., Pentium (2-wide), UltraSPARC & Alpha 21164 (4-wide)
- ❑ **Very Long Instruction Word (VLIW)**
 - + Software figures out what can be done in parallel, so the hardware can be dumb and low power
 - Compiler must group parallel instr's, 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)

2. **Dynamically-scheduled (out-of-order) SuperScalar**

- ❑ Can extract much more ILP with out-of-order processing
- ❑ Pentium Pro/II/III (3-wide), Alpha 21264 (4-wide)

A multiple-issue pipeline



- ❑ Overcome this Flynn bottleneck 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 2, AMD Opteron)
 - ❑ Some more (Power5 is 5-wide; Itanium is 6-wide)
 - ❑ Some less (2-wide is common for simple (low power) cores)

Superscalar pipeline diagrams - Ideal

scalar

lw 0(\$1) → \$2

lw 4(\$1) → \$3

lw 8(\$1) → \$4

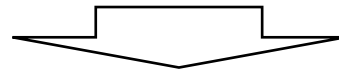
add \$14,\$15 → \$6

add \$12,\$13 → \$7

add \$17,\$16 → \$8

lw 0(\$18) → \$9

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(\$1) → \$2	F	D	X	M	W							
lw 4(\$1) → \$3		F	D	X	M	W						
lw 8(\$1) → \$4			F	D	X	M	W					
add \$14,\$15 → \$6				F	D	X	M	W				
add \$12,\$13 → \$7					F	D	X	M	W			
add \$17,\$16 → \$8						F	D	X	M	W		
lw 0(\$18) → \$9							F	D	X	M	W	



2-way superscalar

lw 0(\$1) → \$2

lw 4(\$1) → \$3

lw 8(\$1) → \$4

add \$14,\$15 → \$6

add \$12,\$13 → \$7

add \$17,\$16 → \$8

lw 0(\$18) → \$9

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(\$1) → \$2	F	D	X	M	W							
lw 4(\$1) → \$3	F	D	X	M	W							
lw 8(\$1) → \$4		F	D	X	M	W						
add \$14,\$15 → \$6		F	D	X	M	W						
add \$12,\$13 → \$7			F	D	X	M	W					
add \$17,\$16 → \$8			F	D	X	M	W					
lw 0(\$18) → \$9				F	D	X	M	W				

Superscalar pipeline diagrams - Realistic

scalar

lw 0(\$1) → \$2

lw 4(\$1) → \$3

lw 8(\$1) → \$4

add \$4, \$5 → \$6

add \$2, \$3 → \$7

add \$7, \$6 → \$8

lw 0(\$8) → \$9

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(\$1) → \$2	F	D	X	M	W							
lw 4(\$1) → \$3		F	D	X	M	W						
lw 8(\$1) → \$4			F	D	X	M	W					
add \$4, \$5 → \$6				F	d	D	X	M	W			
add \$2, \$3 → \$7					f	F	D	X	M	W		
add \$7, \$6 → \$8							F	D	X	M	W	
lw 0(\$8) → \$9								F	D	X	M	W

2-way superscalar

lw 0(\$1) → \$2

lw 4(\$1) → \$3

lw 8(\$1) → \$4

add \$4, \$5 → \$6

add \$2, \$3 → \$7

add \$7, \$6 → \$8

lw 0(\$8) → \$9

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(\$1) → \$2	F	D	X	M	W							
lw 4(\$1) → \$3	F	D	X	M	W							
lw 8(\$1) → \$4		F	D	X	M	W						
add \$4, \$5 → \$6		F	d	d	D	X	M	W				
add \$2, \$3 → \$7			f	f	F	D	X	M	W			
add \$7, \$6 → \$8			f	f	F	D	X	M	W			
lw 0(\$8) → \$9						F	D	X	M	W		

Superscalar CPI calculations

- ❑ Base CPI for scalar pipeline is 1 whereas the **base CPI for N-way superscalar pipeline is $1/N$**

- Why we use IPC instead of CPI; assumes no data hazard or branch stalls (an overly optimistic assumption)
 - Amplifies stall penalties

- ❑ Example: Branch penalty calculation

- ❑ 20% branches, 75% taken, no branch prediction, 2 stall penalty

- ❑ Scalar pipeline

$$1 + 0.2 * 0.75 * 2 = 1.3 \rightarrow 1.3 / 1 = 1.3 \rightarrow 30\% \text{ slowdown}$$

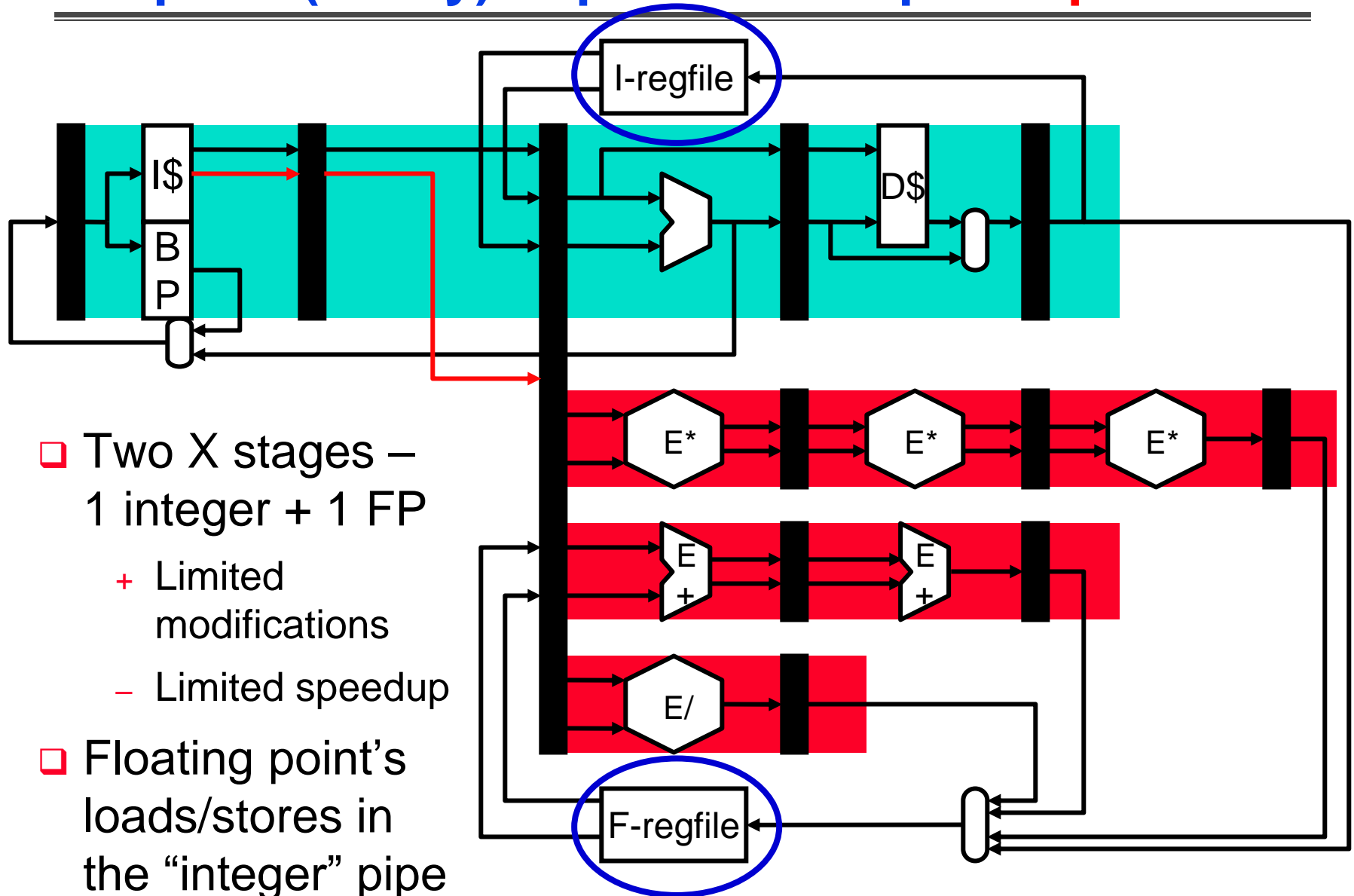
- ❑ 2-way superscalar pipeline

$$\mathbf{0.5} + 0.2 * 0.75 * 2 = 0.8 \rightarrow 0.8 / 0.5 = 1.6 \rightarrow 60\% \text{ slowdown}$$

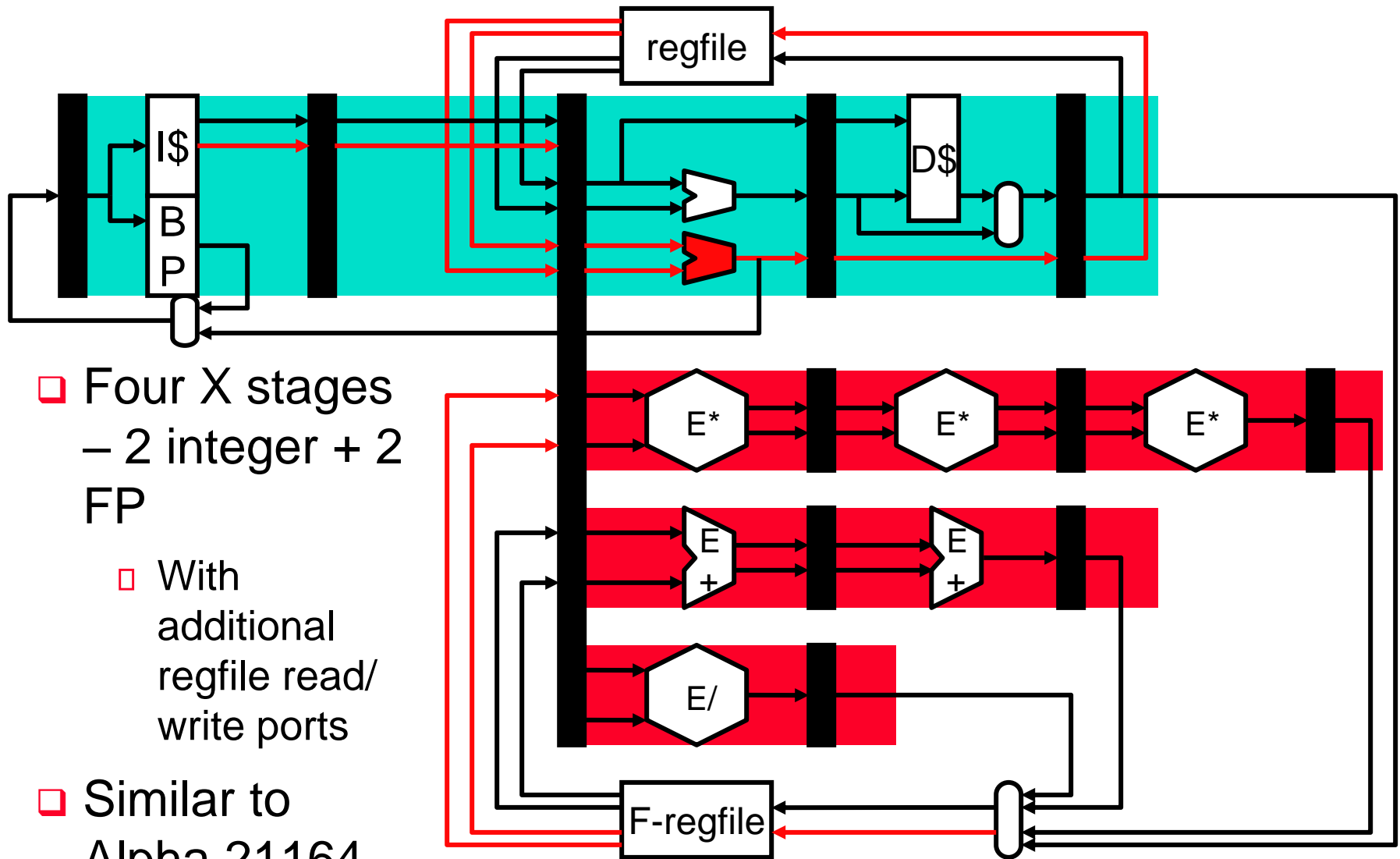
- ❑ 4-way superscalar

$$\mathbf{0.25} + 0.2 * 0.75 * 2 = 0.55 \rightarrow 0.55 / 0.25 = 2.2 \rightarrow 120\% \text{ slowdown}$$

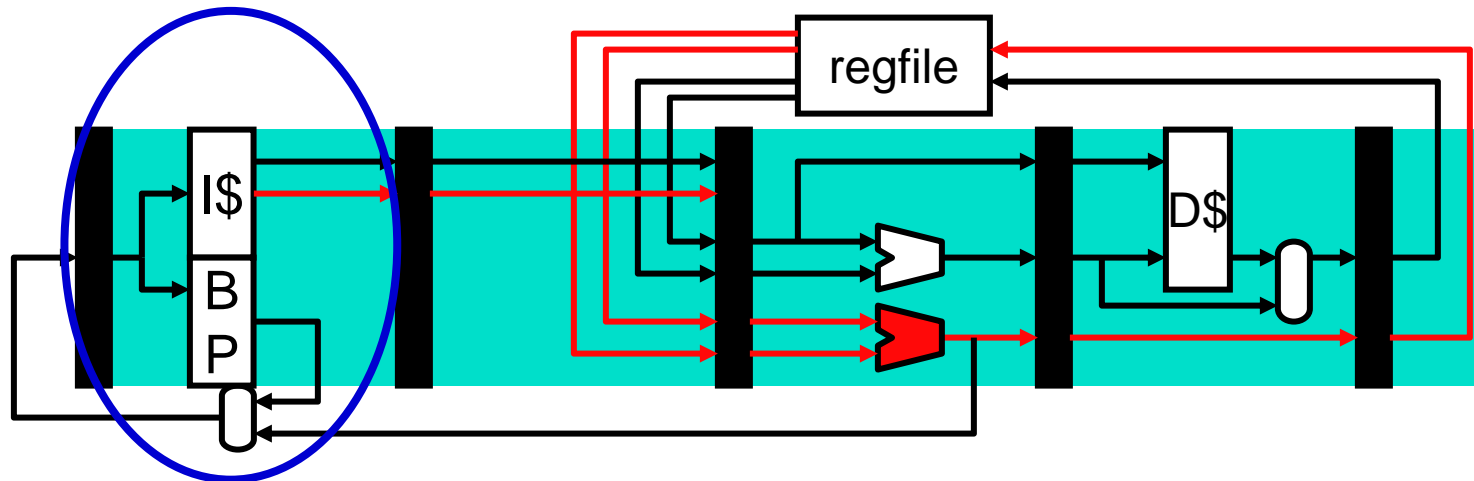
Simplest (2-way) superscalar: Split flt point



A 4-way pipeline (2 integer, 2 FP)

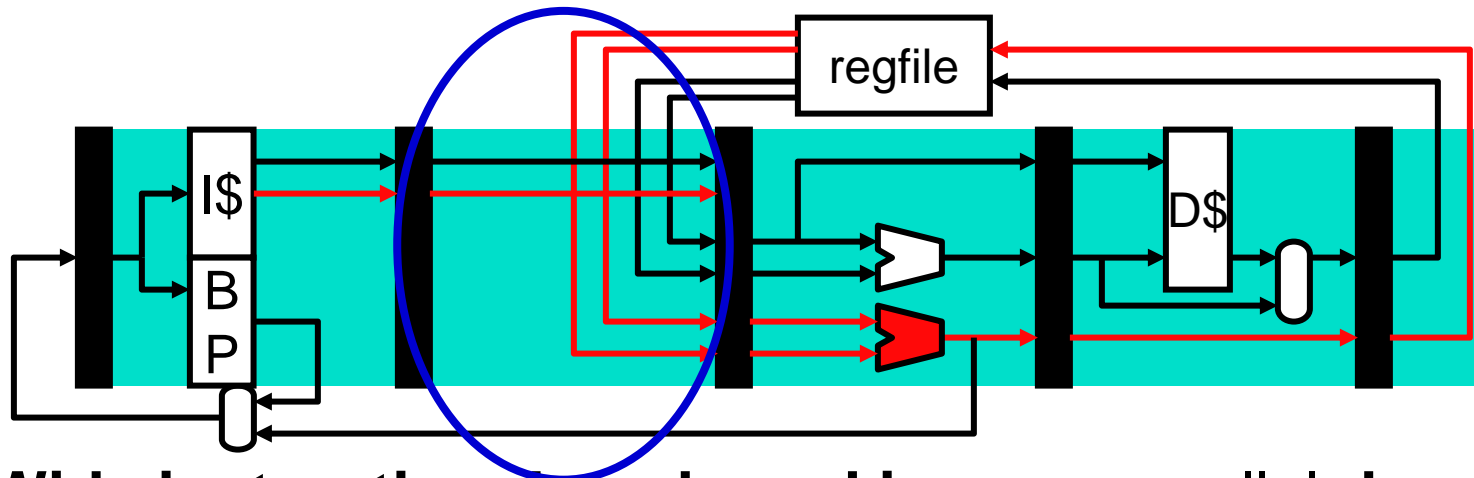


Superscalar challenges – Front end



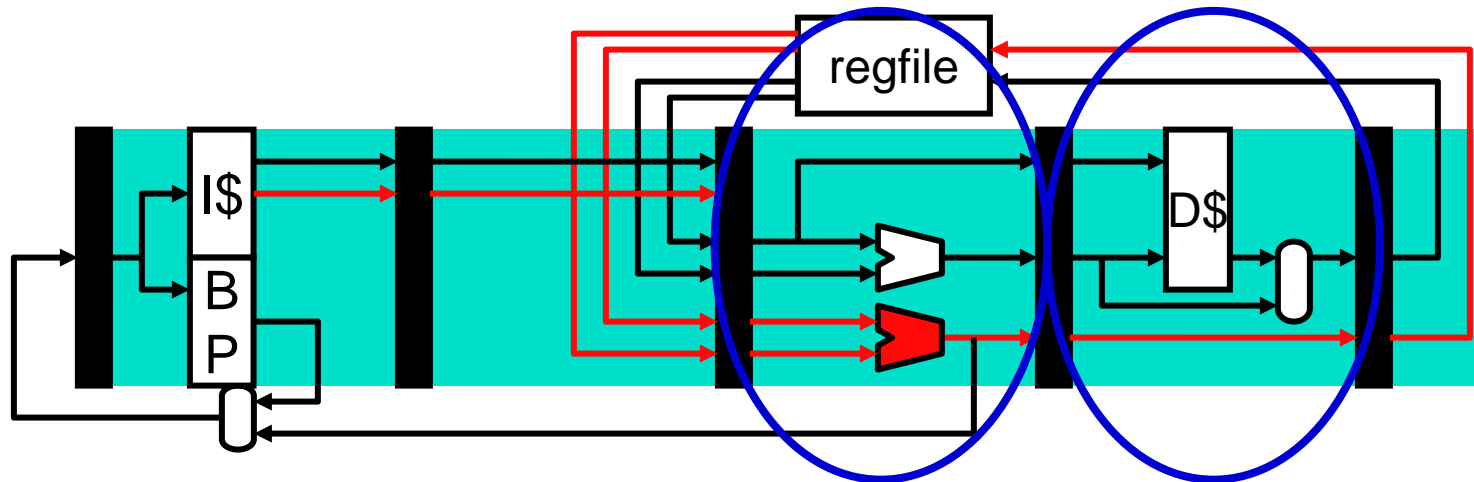
- ❑ **Wide instruction fetch** – Fetch an entire 16B or 32B cache block (4 to 8 instructions (assuming 4-byte instr's))
 - ❑ Need to get from L1 I\$ multiple instructions per cycle (as long as the fetch width doesn't exceed the L1 I\$ size)
 - ❑ Branch prediction ??
 - Modest: predict a **single** branch per fetch cycle
 - Aggressive: predict **multiple** branches per fetch cycle

Superscalar challenges – Front end, con't



- ❑ **Wide instruction decode and issue** – parallel **decode** of 4 (to 8) instr's and decide which can **issue**
 - ❑ Replicate decoders ... and
 - ❑ Determine when instructions can proceed (issue) in parallel – more complex stall logic (order N^2 for N -wide machine)
 - Need to check for conflicting instructions, e.g., instr's that will require the same X stage functional unit in the next cycle
 - Load-use delays and if output of I_1 is an input to I_2
 - ❑ Multi-ported register file – $N*2$ read ports (and N write ports)
 - Larger area, latency, power, cost, complexity

Superscalar challenges – Back end



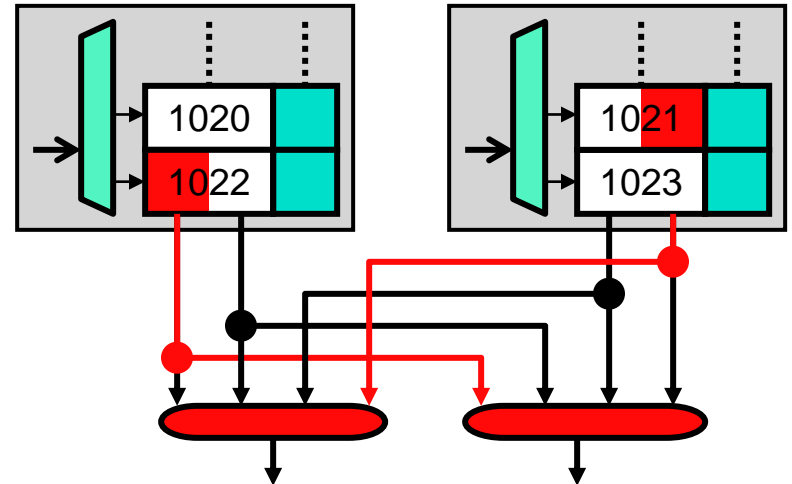
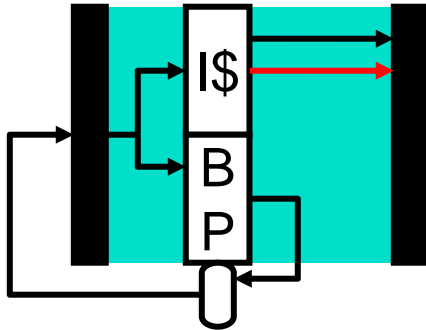
❑ Multiple execution units

- ❑ Simple adders are easy, but bypass paths get expensive
 - More possible sources for data values
 - Order ($N^2 * P$) for N -way machine with execute pipeline depth P

❑ Memory unit

- ❑ Single load per cycle (stall one issue at decode if not) probably okay for dual issue
- ❑ Alternative: add another read port to data cache
 - Larger area, latency, power, cost, complexity

Wide fetch: Sequential instructions



- ❑ What is involved in fetching multiple instructions per cycle?
 - ❑ If in same cache block? → no problem
 - Favors larger block size (independent of hit rate)
 - ❑ Compilers align **basic blocks** to I\$ lines (`.align` directive)
 - Reduces I\$ capacity
 - + Increases fetch bandwidth utilization (more important)
 - ❑ What if fetch packet is 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 fetch: Non-sequential instructions

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

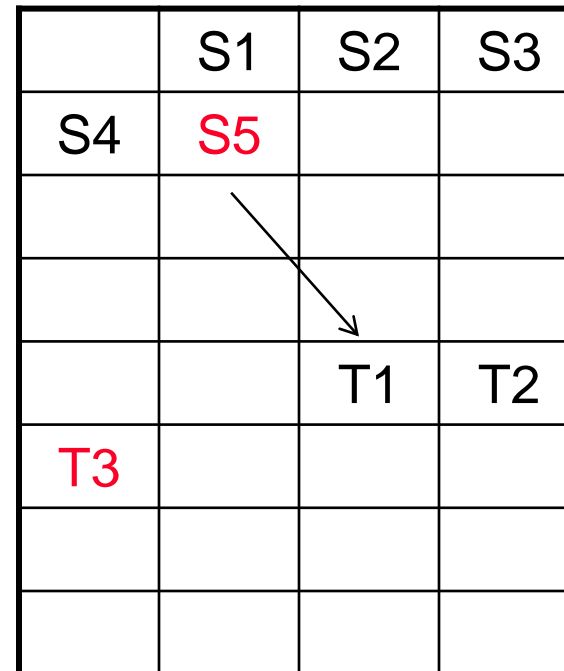
- ❑ Simplest, most common organization: “1” and “No”
 - ❑ One prediction, discard post-branch instr’s if prediction is “taken”
 - Lowers effective fetch width and IPC
 - ❑ Average number of instructions per taken branch?
 - Assume: 20% branches, 50% taken → ~10 instructions
 - ❑ Consider a 10-instruction 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)

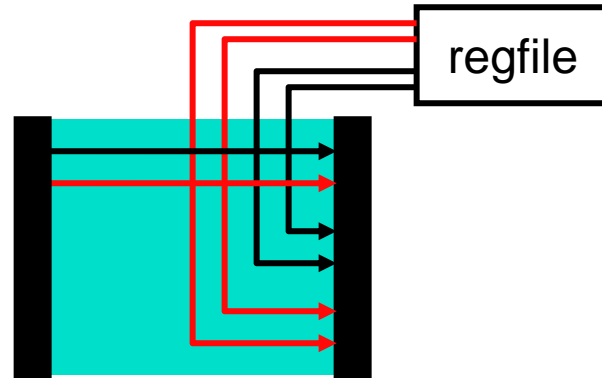
Example of a non-sequential fetch sequence

- ❑ Instruction run – number of instructions (run length) fetched between **taken** branches (**S5** and **T3**)
 - ❑ Instruction fetcher operates most efficiently when processing long runs – unfortunately runs are usually quite short (about six instructions)
- ❑ For a 4-way fetcher, get an instruction bandwidth of only 2 instructions per cycle (**with** branch prediction support)
 - ❑ 8 instructions in 4 cycles

	S1	S2	S3
S4	S5		
		T1	T2
T3			



Wide decode



- ❑ What is involved in decoding multiple (N) instr's per cycle?
- ❑ Actually doing the decoding?
 - ❑ Easier 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**?

N² dependence cross-check

❑ Stall logic for 1-way pipeline with full bypassing

- ❑ Full bypassing → load/use stalls only

$X/M.op == \text{LOAD} \ \&\& \ (D/X.rs1 == X/M.rd \ || \ D/X.rs2 == X/M.rd)$

- ❑ Two “terms”: $\propto 2*N$

❑ Stall logic (load-use) for a 2-way pipeline

$X/M_1.op == \text{LOAD} \ \&\& \ (D/X_1.rs1 == X/M_1.rd \ || \ D/X_1.rs2 == X/M_1.rd) \ ||$

$X/M_1.op == \text{LOAD} \ \&\& \ (D/X_2.rs1 == X/M_1.rd \ || \ D/X_2.rs2 == X/M_1.rd) \ ||$

$X/M_2.op == \text{LOAD} \ \&\& \ (D/X_1.rs1 == X/M_2.rd \ || \ D/X_1.rs2 == X/M_2.rd) \ ||$

$X/M_2.op == \text{LOAD} \ \&\& \ (D/X_2.rs1 == X/M_2.rd \ || \ D/X_2.rs2 == X/M_2.rd)$

- ❑ Eight “terms”: $\propto 2*N^2$

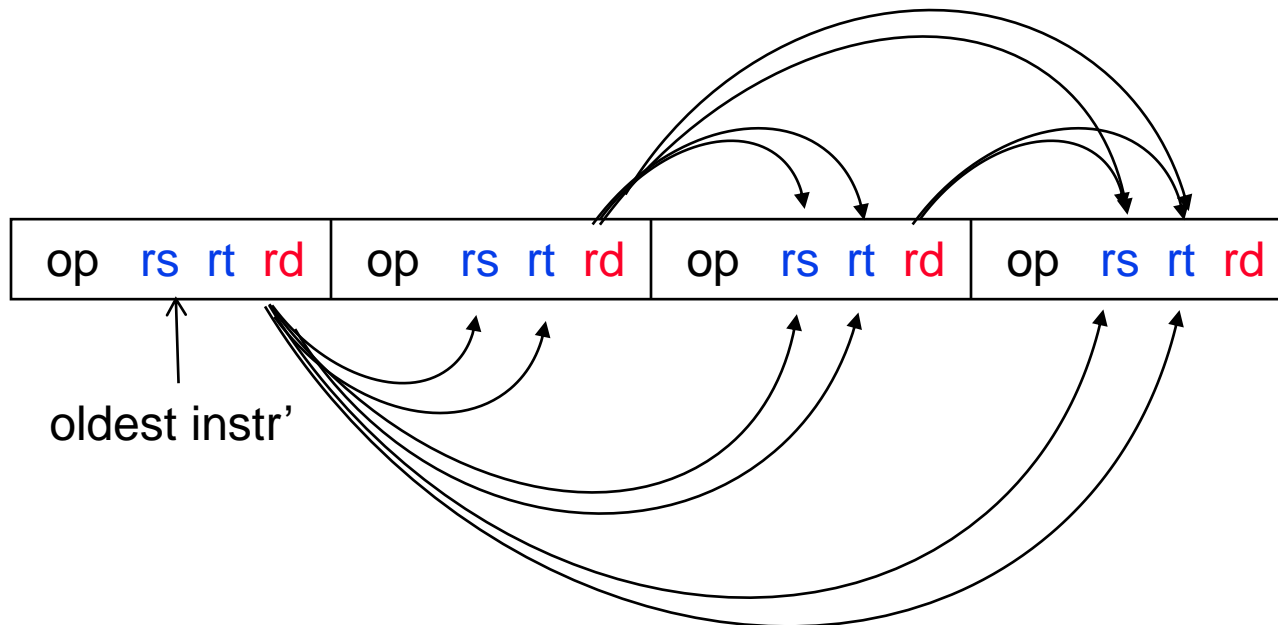
- **N² load-use stall dependence cross-check**

❑ Not quite done, also need “within the fetch bundle” dst- src dependence cross-check

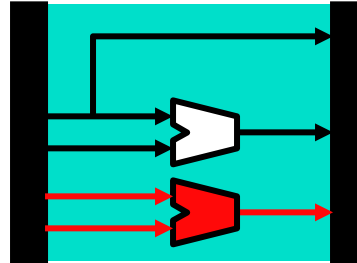
- $D/X_2.rs1 == D/X_1.rd \ || \ D/X_2.rs2 == D/X_1.rd$

4-way dependence cross-checking

- ❑ Check for structural hazards (need same FU's in X ?)
- ❑ Cross check for load-use hazards of the 4 instr's in D/X (both **src** operands) to the 4 instr's in X/M which gives $2 \cdot 4^2$ or 32 dependency checks
- ❑ And check for $\frac{12}{12}$ **dst-src** dependencies between the 4 instructions in the instruction "bundle"

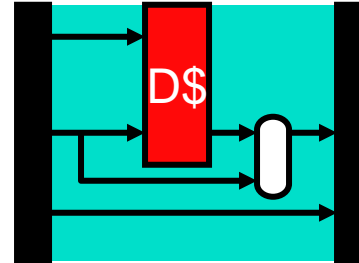


Wide execute



- ❑ What is involved in executing N instr's per cycle?
- ❑ Multiple execution units ... N of every kind?
 - ❑ N ALUs? OK, ALUs are small
 - ❑ N FP dividers? No, FP dividers are huge and `fdi` is uncommon
 - ❑ How many branches per cycle? How many loads/stores per cycle?
 - ❑ Typically some mix of functional units proportional to instr mix
 - Intel Pentium: 1 any + 1 ALU
 - Alpha 21164: 2 integer (including 2 loads) + 2 FP

Wide memory access



- ❑ What about multiple loads/stores per cycle?
 - ❑ Probably only necessary on processors 4-wide or wider
 - ❑ More important to support multiple loads than multiple stores
 - Instr mix: loads (~20–25%), stores (~10–15%)
 - ❑ Alpha 21164: two loads *or* one store per cycle

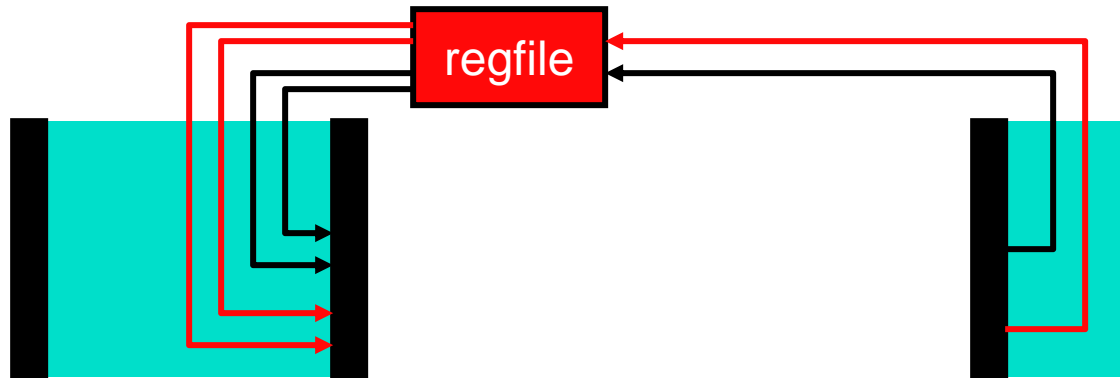
D\$ bandwidth: Multi-porting, replication

- ❑ 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 two loads, no latency penalty
 - Not a solution for two stores (probably that's OK), area (cost), power penalty
 - Is this what Alpha 21164 does?

D\$ bandwidth: Banking

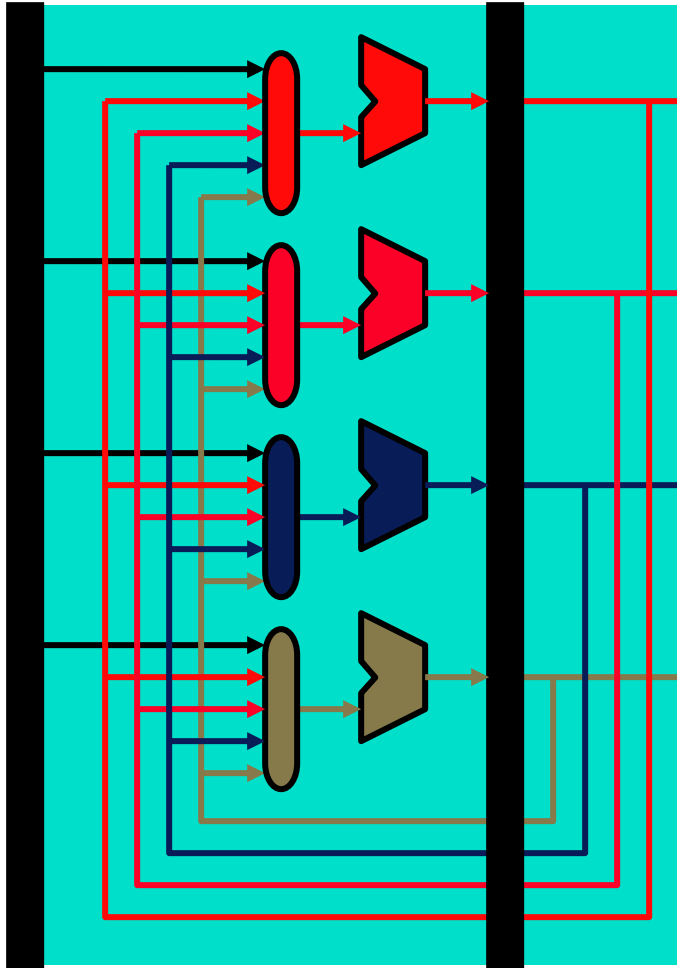
- ❑ Option#3: **banking** (or **interleaving**)
 - ❑ Divide D\$ into “banks” (by address), 1 access/bank-cycle
 - ❑ **Bank conflict**: two accesses to same bank → one stalls
 - + No latency, area, power overheads (latency may even be lower)
 - + One access per bank per cycle, **assuming no conflicts**
 - Complex stall logic → address not known until execute stage
 - To support N accesses, need $2N+$ banks to avoid frequent conflicts
- ❑ Which address bit(s) determine bank?
 - ❑ Offset bits? Individual cache line spread across different banks
 - + Fewer conflicts
 - Must replicate tags across banks, complex miss handling
 - ❑ Index bits? Banks contain complete cache lines
 - More conflicts
 - + Tags not replicated, simpler miss handling

Wide register read/write



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

Wide 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
 - Big capacitive load
- And this is just one bypass stage (MX)!
- There is also WX and WM bypassing
 - Even more for deeper pipelines
- One of the big problems of superscalar

□ Implemented as bit-slicing

- 64 1-bit bypass networks
- Mitigates routing problem somewhat

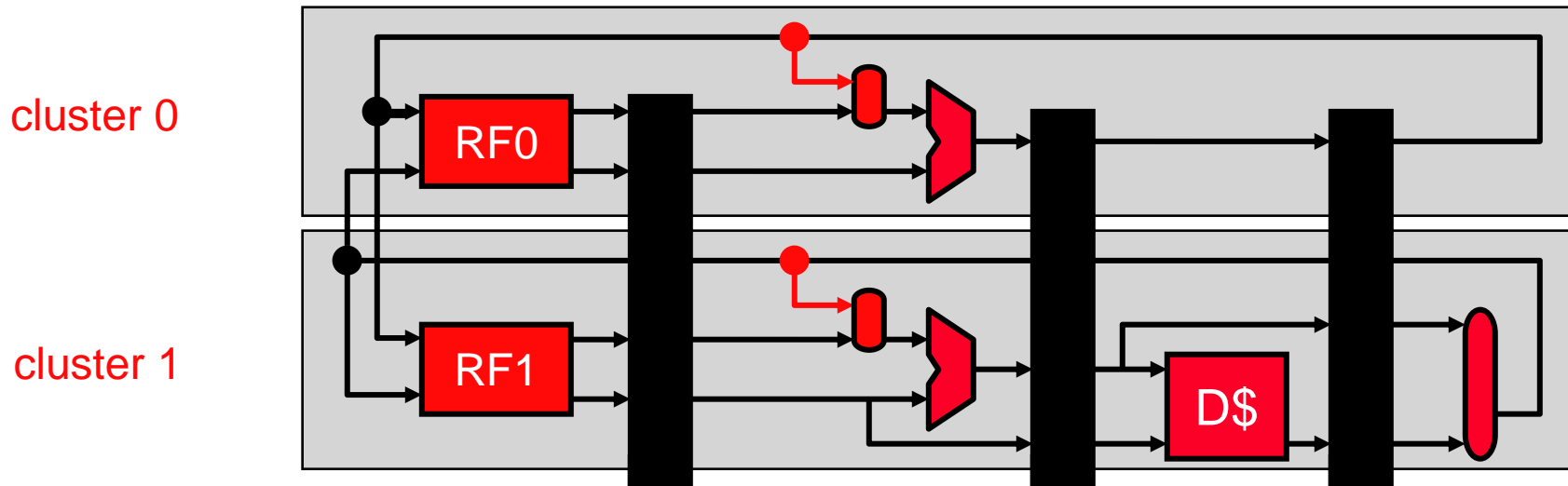
Not all N^2 created equal

- ❑ N^2 bypass vs. N^2 stall logic & dependence cross-check
 - ❑ Which is the bigger problem?

- ❑ N^2 bypass ... by far
 - ❑ 32- or 64- bit quantities (vs. 5-bit register addresses)
 - ❑ 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 is also more serious than cross-check

Avoid N^2 bypass/RegFile: Clustering



- ❑ **Clustering**: group ALUs into **K** clusters
 - ❑ Full bypassing within cluster, limited (or no) bypassing between them
 - Get values from regfile with 1 or 2 cycle delay
 - + N/K non-regfile inputs at each mux, N^2/K point-to-point paths
 - ❑ Key to perf: hardware steers dependent instr's to same cluster
 - ❑ Hurts IPC, but helps clock frequency (or wider issue at same clock)
- ❑ Typically used with replicated regfile: replica per cluster
 - ❑ Alpha 21264: 4-way superscalar, two clusters

Trends in in-order superscalar processors

	486	Pentium	PentiumII	Pentium4	Itanium	ItaniumII	Core2
Year	1989	1993	1998	2001	2002	2004	2006
Width	1	2	3	3	3	6	4

- ❑ Issue width saturated at 4-6 for high-performance cores for many years
 - ❑ Canceled Alpha 21464 was 8-way issue
 - ❑ Little justification for going wider
 - ❑ Hardware or compiler “scheduling” needed to exploit 4-6 effectively
 - Out-of-order execution (or VLIW/EPIC)
- ❑ For high-performance **per watt** cores, issue width is ~2
 - ❑ Advanced scheduling techniques not needed
 - ❑ Multi-threading (a little later) helps cope with cache misses

- ❑ So far we have only considered 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 instr word (VLIW)**
 - ❑ Effectively, a 1-wide pipeline, but each “fetch unit” is an N-instr group
 - ❑ Compiler guarantees instr’s within a VLIW group are independent
 - If no independent instr’s, 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 instr must be ALU, 2nd mem, etc.
 - + Further simplification

History of VLIW

- ❑ Started with “horizontal microcode”
- ❑ Academic projects
 - ❑ Yale ELI-512 [Fisher, '85]
 - ❑ Illinois IMPACT [Hwu, '91]
- ❑ Commercial attempts
 - ❑ Multiflow [Colwell+Fisher, '85] → failed
 - ❑ Cydrome [Rau, '85] → failed
 - ❑ Motorola/TI embedded processors → successful
 - ❑ Intel Itanium [Fisher+Rau, '97] → ??
 - ❑ Transmeta Crusoe [Ditzel, '99] → mostly failed
 - ❑ Nvidia Denver [~2014] → ongoing

Pure and “tainted” VLIW

- ❑ **Pure VLIW**: no hardware dependence checks at all
 - ❑ Not even between VLIW groups
 - + Very simple and low power hardware
 - ❑ Compiler responsible for **scheduling** stall cycles
 - ❑ Requires precise knowledge of pipeline depth and structure
 - These must be fixed for compatibility
 - Doesn't support caches (i.e., cache misses) well
 - ❑ Used in some cache-less micro-controllers, but not generally useful
- ❑ **Tainted (more realistic) VLIW**: inter-group checks
 - ❑ Compiler doesn't schedule stall cycles
 - + Precise pipeline depth and latencies not needed, can be changed
 - + Supports caches
 - ❑ TransMeta Crusoe

What does VLIW actually buy you?

- + Simpler I\$/branch prediction
- + 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

❑ EPIC (Explicitly Parallel Instr Computing)

- ❑ “New” VLIW (Variable Length Instr Words)
- ❑ Implemented as “bundles” with explicit dependence bits
- ❑ Code is compatible with different “bundle” width machines
- ❑ Compiler discovers as much parallelism as it can, hardware does rest
- ❑ E.g., Intel Itanium (IA-64)
 - 128-bit bundles (three 41-bit instr’s + 4 dependence bits)

❑ Still does not address bypassing or register file issues

Aside: Instruction scheduling

- ❑ Idea: place independent instr's between slow ops and load-uses
 - ❑ Otherwise, pipeline stalls while waiting for RAW hazards to resolve
- ❑ To schedule well need ... **independent instr's**
- ❑ **Scheduling scope**: code region we are scheduling
 - ❑ The bigger the better (more independent instr's to choose from)
 - A large scope means scheduling across branches
 - ❑ Once scope is defined, schedule is pretty obvious
- ❑ Compiler scheduling (really scope enlarging) techniques
 - ❑ Loop unrolling (for loops)
 - + Simple
 - Expands code size, can't handle recurrences or non-loops
 - ❑ Trace scheduling (for non-loop control flow)

Aside: Scheduling - Compiler or hardware ?

❑ Compiler

- + Potentially large scheduling scope (full program)
- + Simple hardware → fast clock, short pipeline, and low power
- Low branch prediction accuracy (profiling?)
- Little information on memory dependences (profiling?)
- Can't dynamically respond to cache misses
- Pain to speculate and recover from mis-speculation (h/w support?)

❑ Hardware

- + High branch prediction accuracy
- + Dynamic information about memory dependences
- + Can respond to cache misses
- + Easy to speculate and recover from mis-speculation
- Finite buffering resources fundamentally limit scheduling scope
- Scheduling machinery adds pipeline stages and consumes power

Multiple issue redux

❑ Multiple issue

- ❑ Needed to expose instr level parallelism (ILP) beyond pipelining
- ❑ Improves performance, but reduces utilization
- ❑ 4-6 way issue is about the peak issue width currently justifiable

❑ Problem spots

- ❑ Fetch + branch prediction → trace cache?
- ❑ N^2 bypass → clustering?
- ❑ N^2 register file read ports → replication?

❑ Compiler can help extract parallelism - “schedule” code to avoid stalls

- ❑ Important for (in order) superscalar
- ❑ Critical for VLIW/EPIC