

CSE 560 Computer Systems Architecture

Superscalar

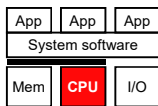
1

Remainder of CSE 560: Parallelism

- Last unit: **pipeline-level parallelism**
 - Execute one instruction in parallel with decode of next
- Next: **instruction-level parallelism (ILP)**
 - Execute multiple independent instructions fully in parallel
 - Today: multiple issue
 - In a few weeks: dynamic scheduling
 - Extract much more ILP via out-of-order processing
- Data-level parallelism (DLP)**
 - Single-instruction, multiple data
 - Ex: one instruction, four 16-bit adds (using 64-bit registers)
- Thread-level parallelism (TLP)**
 - Multiple software threads running on multiple cores

6

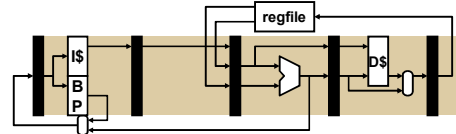
This Unit: Superscalar Execution



- 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)

7

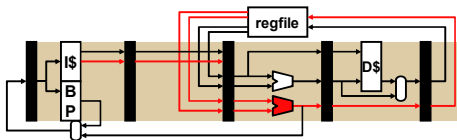
Scalar Pipeline and the Flynn Bottleneck



- So far we have looked at **scalar pipelines**:
 - 1 instruction per stage (+ 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)

8

Multiple-Issue Pipeline



- Overcome this limit using **multiple issue**
 - Also called **superscalar**
 - Two instructions per stage at once (or 3 or 4 or 8...)
 - "Instruction-Level Parallelism (ILP)"** [Fisher, IEEE TC'81]
- Today, typically "4-wide" (Intel Core 2, AMD Opteron)
 - Some more (Power5 is 5-issue; Itanium is 6-issue)
 - Some less (dual-issue is common for simple cores)

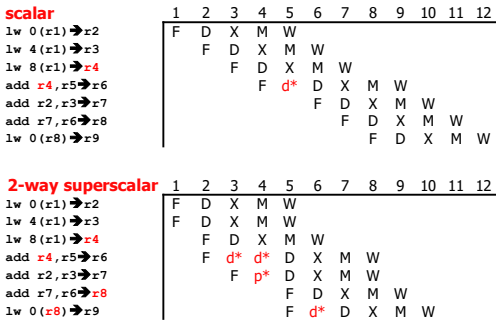
9

Superscalar Pipeline Diagrams - Ideal

	1	2	3	4	5	6	7	8	9	10	11	12
scalar												
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	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	

10

Superscalar Pipeline Diagrams - Realistic



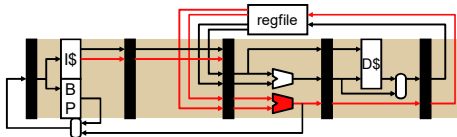
11

Superscalar CPI Calculations

- 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, no explicit branch prediction
- Scalar pipeline
 - $1 + 0.2 \times 0.75 \times 2 = 1.3 \rightarrow 1.3/1 = 1.3 \rightarrow 30\% \text{ slowdown}$
- 2-way superscalar pipeline
 - $0.5 + 0.2 \times 0.75 \times 2 = 0.8 \rightarrow 0.8/0.5 = 1.6 \rightarrow 60\% \text{ slowdown}$
- 4-way superscalar
 - $0.25 + 0.2 \times 0.75 \times 2 = 0.55 \rightarrow 0.55/0.25 = 2.2 \rightarrow 120\% \text{ slowdown}$

12

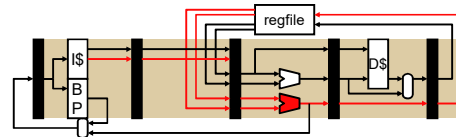
A Typical Dual-Issue Pipeline (1)



- 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)

13

A Typical Dual-Issue Pipeline (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
 - 1 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

14

Superscalar Challenges - Front End

- Wide instruction fetch**
 - Modest: need multiple instructions per cycle
 - Aggressive: predict multiple branches
- Wide instruction decode**
 - Replicate decoders
- Wide instruction issue**
 - 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 \rightarrow 8 read ports

15

Superscalar Challenges - Back End

- Wide instruction execution**
 - Replicate arithmetic units
 - Perhaps multiple cache ports
- Wide bypass paths**
 - More possible sources for data values
 - Order $(N^2 \times P)$ for N -wide machine, execute pipeline depth P
- Wide 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

16

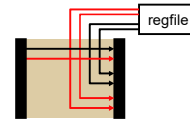
How Much ILP is There?

- The compiler tries to "schedule" code to avoid stalls
 - Hard for scalar machines (to fill load-use delay slot)
 - Even harder to schedule multiple-issue (superscalar)
- Even given unbounded ILP, superscalar has limits
 - IPC (or CPI) vs clock frequency trade-off
 - Given these challenges, what is reasonable N? 3 or 4 today

17

17

Wide Decode



- What is involved in decoding multiple (N) insns per cycle?
- Actually doing the decoding?
 - Easy if fixed length (multiple decoders), doable if variable
- Reading input registers?
 - 2N register read ports (latency \propto #ports)
 - Actually $< 2N$, most values come from bypasses (more later)
- What about the **stall logic**?

20

20

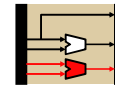
N^2 Dependence Cross-Check

- Stall logic for 1-wide pipeline with full bypassing
 - Full bypassing \rightarrow load/use stalls only
 - $X/M.op == \text{LOAD} \ \&\& \ (D/X.rs1 == X/M.rd \ || \ D/X.rs2 == X/M.rd)$
 - Two "terms": $\propto 2N$
- Now: same logic for a 2-wide 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": $\propto 2N^2$
 - N^2 dependence cross-check**
 - Not quite done, also need
 - $D/X_2.rs1 == D/X_1.rd \ || \ D/X_2.rs2 == D/X_1.rd$

21

21

Wide Execute

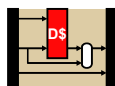


- What is involved in executing N insns per cycle?
- Multiple execution units ... N of every kind?
 - N ALUs? OK, ALUs are small
 - N FP dividers? No, FP dividers are huge, `fdiv` is uncommon
 - How many branches/cycle? How many loads/stores /cycle?
- Typically mix of functional units proportional to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

22

22

Wide Memory Access

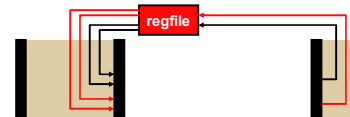


- What about multiple loads/stores per cycle?
 - Probably only necessary on processors 4-wide or wider
 - More important to support multiple loads than stores
 - Insn mix: loads (~20–25%), stores (~10–15%)
 - Alpha 21164: two loads *or* one store per cycle

23

23

Wide Register Read/Write

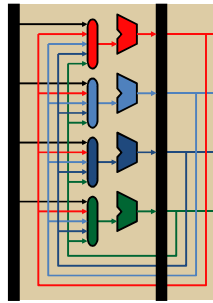


- How many register file ports to execute N insns per cycle?
 - Nominally, 2N read + N write (2 read + 1 write per insn)
 - Latency, area \propto #ports²
 - In reality, fewer than that
 - Read ports: many values from bypass network, immediates
 - Write ports: stores, branches (35% insns) don't write registers
- Replication works great for regfiles (used in Alpha 21164)

24

24

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
- 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
 - Implemented as bit-slicing
 - 64 1-bit bypass networks
 - Mitigates routing problem somewhat

25

25

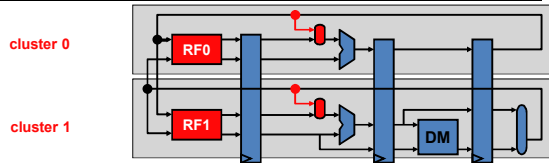
Not All N^2 Created Equal

- 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)
 - Multiple bypass levels (MX, WX) 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 also N^2 problem (think latency where N is #ports)
 - And also more serious than cross-check

26

26

Avoid N^2 Bypass/RegFile: Clustering



- Clustering:** group ALUs into K clusters
- Full bypassing within cluster, limited bypassing between clusters
 - Get values from regfile with 1-2 cycle delay
 - + N/K non-regfile inputs at each mux, N^2/K point-to-point paths
 - Key to performance: **steering** dependent insns to same cluster
 - Hurts IPC, helps clock frequency (or wider issue at same clock)
 - Typically uses replicated register files (1 per cluster)
 - Alpha 21264: 4-way superscalar, two clusters

27

27

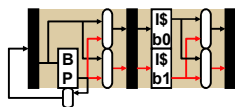
Wide Non-Sequential Fetch

- **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"**
 - 1 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 $\rightarrow \sim 10$ instructions
 - Consider: 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)

28

28

Parallel Non-Sequential Fetch



- Allowing "embedded" taken branches is possible
 - Requires smart branch predictor, multiple I\$ accesses/cycle
- Can try pipelining branch prediction and fetch
 - Branch prediction stage only needs PC
 - Transmits two PCs to fetch stage, next PC and next-next PC
 - Elongates pipeline, increases branch penalty
 - Pentium II & III do something like this
- Another option: loop cache

29

29

Multiple-issue CISC

- How do we apply superscalar techniques to CISC?
- Break "macro-ops" into "micro-ops"
 - Also called "μops" or "RISC-ops"
 - A typical CISCy instruction "add [r1], [r2] \rightarrow [r3]" becomes:
 - Load [r1] \rightarrow t1 (t1 = temp. register, not visible to sw)
 - Load [r2] \rightarrow t2
 - Add t1, t2 \rightarrow t3
 - Store t3 \rightarrow [r3]
 - Internal pipeline manipulates only RISC-like instructions
- But, conversion can be expensive (latency, area, power)
 - Solution: cache converted instructions in **trace cache**

30

30

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)
- We've already talked about statically-scheduled superscalar

31

31

VLIW

- Hardware-centric multiple issue problems:
 - Wide fetch+ br. prediction, N^2 bypass, N^2 dependence checks
 - HW solutions: 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 group are independent
 - Gaps filled with `nops`
 - Group travels down pipeline as a unit
 - + Simplifies pipeline control (no rigid vs. fluid business)
 - + Cross-checks within a group unnecessary
 - Downstream cross-checks still necessary
 - Typically "slotted": 1st insn must be ALU, 2nd mem, etc.
 - + Further simplification

32

32

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

33

33

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

34

34

Trends in Single-Processor Multiple Issue

	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
 - Canceled Alpha 21464 was 8-way issue
 - No justification for going wider
 - HW 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

37

37