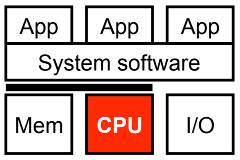
CIS 371
Computer Organization and Design

Unit 6: Superscalar Pipelines

A Key Theme of CIS 371: Parallelism

- Last unit: 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
  - Today: multiple issue
- Later:
  - 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

This Unit: (In-Order) Superscalar Pipelines

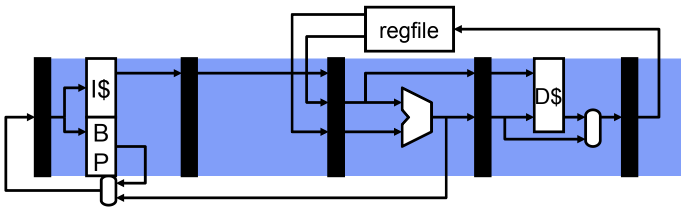


- Superscalar hardware issues
  - Bypassing and register file
  - Stall logic
  - Fetch and branch prediction
- Multiple-issue designs
  - "Superscalar"
  - VLIW/EPIC

Readings

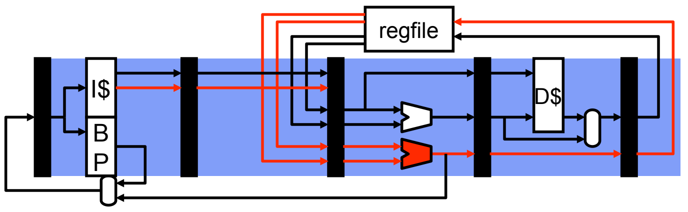
- P&H
  - Chapter 4.10

Scalar Pipeline and the Flynn Bottleneck



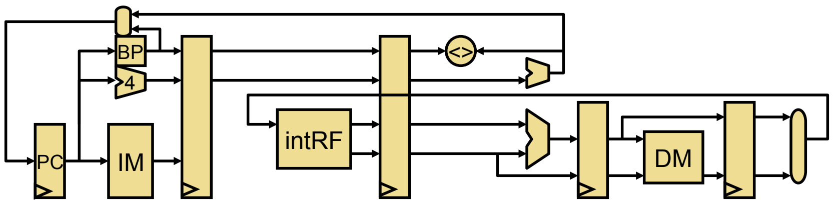
- So far we have looked at **scalar pipelines**
  - One instruction per stage
    - With control speculation, bypassing, etc.
  - – Performance limit (aka “Flynn Bottleneck”) is CPI = IPC = 1
  - – Limit is never even achieved (hazards)
  - – Diminishing returns from “super-pipelining” (hazards + overhead)

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 2, AMD Opteron)
  - Some more (Power5 is 5-issue; Itanium is 6-issue)
  - Some less (dual-issue is common for simple cores)

Scalar Pipelines



- So far we have looked at **scalar pipelines**
  - One instruction per stage

  - With control speculation
  - With bypassing (not shown)
  - With floating-point …

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		

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 optmistic assumption)

- Example: Branch penalty calculation
  - 20% branches, 75% taken, no explicit 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

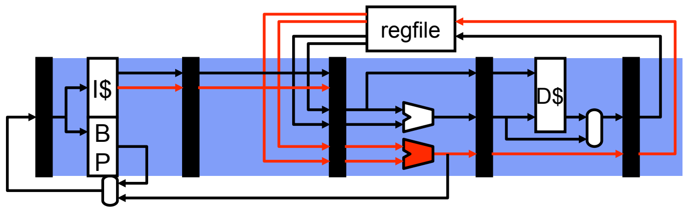
How Much ILP is There?

- 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

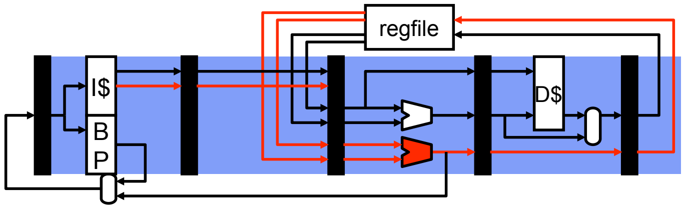
- Even given unbounded ILP, superscalar has limits
  - IPC (or CPI) vs clock frequency trade-off

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)

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 Execution

- Common design: functional unit mix ∝ insn type mix
  - Integer apps: 20–30% loads, 10–15% stores, 15–20% branches
  - Floating point apps: 30% FP, 20% loads, 10% stores, 5% branches
  - Rest 40–50% are non-branch integer ALU operations

  - Intel Pentium (2-way superscalar): 1 any + 1 integer ALU
  - Alpha 21164: 2 integer (incl. 2 loads or 1 store) + 2 floating point
- Execution units
  - Simple ALUs are cheap (have N of these for N-wide processor)
  - Complex ALUs are less cheap (have fewer of these)
  - Data memory bandwidth expensive
    - Multi-port, replicate, or ”bank” (more later in memory unit)

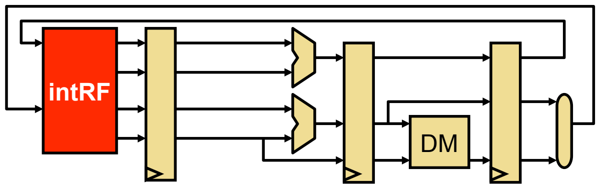
Superscalar Challenges - Front End

- **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 ➔ 8 read ports

Superscalar Challenges - Back End

- **Wide instruction execution**
  - Replicate arithmetic units
  - Perhaps 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 Register File

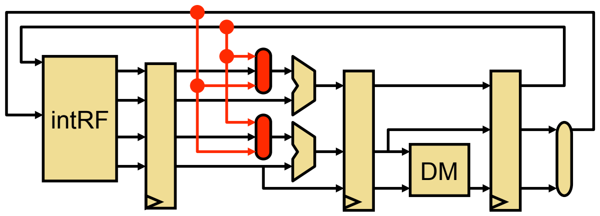


- "N-way superscalar register file: 2N read + N write ports
  - < N write ports: stores, branches (35% insns) don't write registers
  - < 2N read ports: many inputs come from immediates/bypasses
  - – Latency and area ∝ #ports^2 ∝ $(3N)^2$ (slow for large N)

N^2 Dependence Cross-Check

- Stall logic for 1-wide pipeline with full bypassing
  - Full bypassing → load/use stalls only
    X/M.op==LOAD && (D/X.rs1==X/M.rd || D/X.rs2==X/M.rd)
  - Two "terms": ∝ 2N
- 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": ∝ $2N^2$
    - **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 Bypass

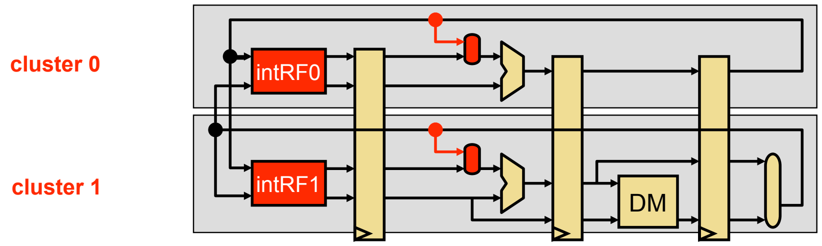


- Consider WX bypass for 1st input of each insn
  - – 2 non-regfile inputs to bypass mux: in general N
  - – 4 point-to-point connections: in general N^2
  - – Bypass wires long (slow) and are difficult to route
  - And this is just one bypass stage and one input per insn!
- **N^2 bypass**

Not All N^2 Problems Created Equal

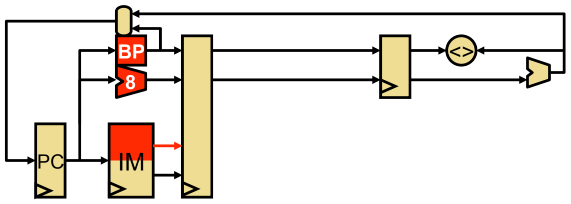
- N^2 bypass vs. N^2 stall logic & dependence cross-check
  - Which is the bigger problem?
- N^2 bypass … by a lot
  - 32- or 64- bit quantities (vs. 5-bit)
  - Multiple levels (MX, WX) of bypass (vs. 1 level of stall logic)
  - Must fit in one clock period with ALU (vs. not)
- Dependence cross-check 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

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 performance: steer dependent insns 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, 2 clusters, static steering

Superscalar Fetch/Decode



- What is involved in fetching N insns per cycle?
  - Mostly wider instruction memory ports
    - Read N instructions in parallel
  - Most tricky aspects involve branch prediction
- What about Decode?
  - Easier with fixed-width instructions (MIPS, Alpha, PowerPC, ARM)
  - Harder with variable-length instructions (x86)

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 → ~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)

Aside: VLIW/EPIC

- **VLIW: Very Long Insn Word**
  - Intel: **EPIC** (Explicit Parallel Instruction Computing)
  - Effectively, a 1-wide pipeline, but unit is an N-insn group
  - Group travels down pipeline as a unit
  - Compiler guarantees insns within a VLIW group are independent
    - If no independent insns, slots filled with `nops`
  - Typically "slotted": 1st insn must be ALU, 2nd mem, etc.
  - E.g., Itanium (two 3-wide bundles per cycle = 6-way issue)
  - + Simplifies fetch and branch prediction
  - + Simplifies pipeline control (no rigid vs. fluid business)
  - – Doesn't help bypasses or regfile, which are bigger problems
    - Can expose these issues to software, too (yuck)
  - – Not really compatible across machines of different widths
    - How does Itanium deal with non-compatibility? Transmeta?

Predication

- Branch mis-predictions hurt more on superscalar
  - Replace difficult branches with something else…
  - Convert control flow into data flow (& dependencies)

- **Predication**
  - Conditionally executed insns unconditionally fetched
  - **Full predication** (ARM, Intel Itanium)
    - Can tag every insn with predicate, but extra bits in instruction
  - **Conditional moves** (Alpha, x86)
    - Construct appearance of full predication from one primitive

```
cmoveq r1,r2,r3        // if (r1==0) r3=r2;
```

    – May require some code duplication to achieve desired effect
    + Only good way of adding predication to an existing ISA
- **If-conversion**: replacing control with predication

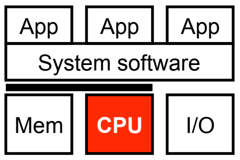
Trends in Single-Processor Multiple Issue

	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 has saturated at 4-6 for high-performance cores
  - Canceled Alpha 21464 was 8-way issue
  - No justification for going wider
  - Hardware or compiler “scheduling” needed to exploit 4-6 effectively

- 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

Multiple Issue Summary



- Superscalar hardware issues
  - Bypassing and register file
  - Stall logic
  - Fetch and branch prediction

- Multiple-issue designs
  - “Superscalar”
  - VLIW

- Next up
  - Midterm