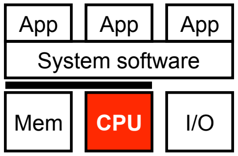
CIS 371
Computer Organization and Design

Unit 9: Superscalar Pipelines

A Key Theme of CIS 371: Parallelism

- Previously: 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

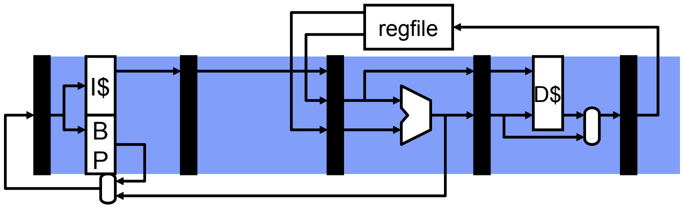


- Idea of instruction-level parallelism
- Superscalar hardware issues
  - Bypassing and register file
  - Stall logic
  - Fetch and branch prediction
- “Superscalar” vs VLIW/EPIC

Readings

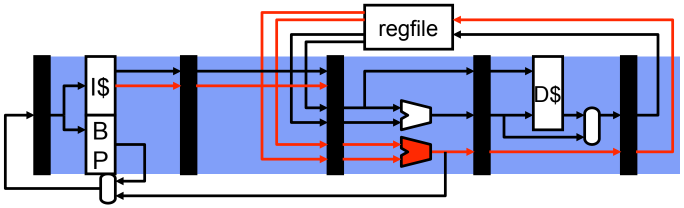
- P&H
  - Chapter 4.10

Scalar Pipeline and the Flynn Bottleneck



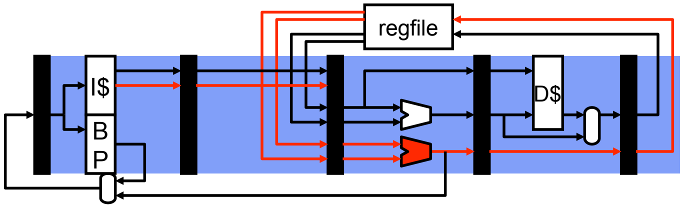
- So far we have looked at **scalar pipelines**
  - One instruction per stage
    - With control speculation, bypassing, etc.
  - Performance limit (aka "Flynn Bottleneck") is CPI = IPC = 1
  - Limit is never even achieved (hazards)
  - Diminishing returns from "super-pipelining" (hazards + overhead)

Multiple-Issue Pipeline



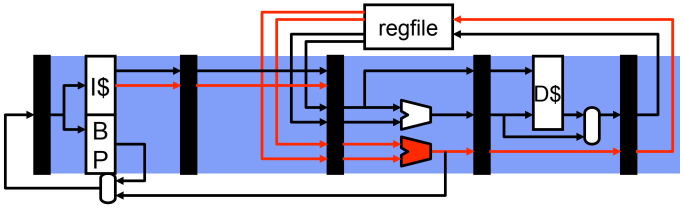
- Overcome this limit using **multiple issue**
  - Also called **superscalar**
  - Two instructions per stage at once, or three, or four, or eight…
  - **"Instruction-Level Parallelism (ILP)"** [Fisher, IEEE TC'81]
- Today, typically "4-wide" (Intel Core i7, AMD Opteron)
  - Some more (Power5 is 5-issue; Itanium is 6-issue)
  - Some less (dual-issue is common for simple cores)

A Typical Dual-Issue Pipeline



- Fetch an entire 16B or 32B cache block
  - 4 to 8 instructions (assuming 4-byte average instruction length)
  - 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 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		

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 Challenges - Front End

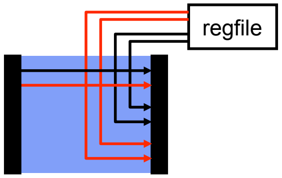
- **Superscalar instruction fetch**
  - Modest: need multiple instructions per cycle
  - Aggressive: predict multiple branches
- **Superscalar instruction decode**
  - Replicate decoders
- **Superscalar instruction issue**
  - Determine when instructions can proceed in parallel
  - Not all combinations possible
  - More complex stall logic - order N^2 for *N*-wide machine
- **Superscalar 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

- **Superscalar instruction execution**
  - Replicate arithmetic units
  - Perhaps multiple cache ports
- **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 ➔ 4 write ports

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

Superscalar Decode & 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?
  - Nominally, 2N read + N write (2 read + 1 write per insn)
    - – Latency, area ∝ #ports^2
- What about the **stall logic**?

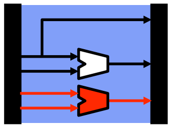
N^2 Dependence Cross-Check

- 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)

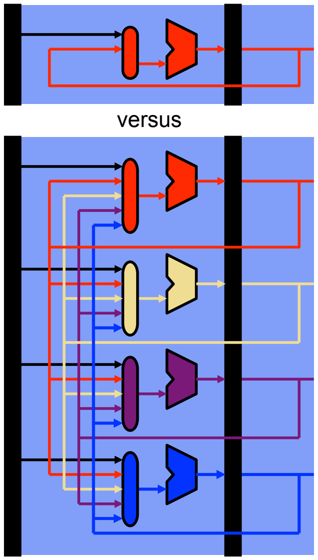
  - Eight "terms": ∝ 2N^2
    - **N^2 dependence cross-check**
  - Not quite done, also need
    - D/X$_2$.rs1==D/X$_1$.rd || D/X$_2$.rs2==D/X$_1$.rd

Superscalar Execute



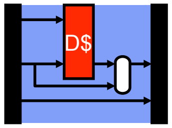
- What is involved in executing N insns per cycle?
- Multiple execution units … N of every kind?
  - N ALUs? OK, ALUs are small
  - N floating point dividers? No, dividers are big, `fdiv` is uncommon
  - How many branches per cycle? How many loads/stores per cycle?
  - Typically some mix of functional units proportional to insn mix
    - Intel Pentium: 1 any + 1 "simple" (such as ADD, etc.)
    - Alpha 21164: 2 integer (including 2 loads) + 2 floating point

Superscalar Bypass



- **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

Superscalar Memory Access



- What about multiple loads/stores per cycle?
  - Probably only necessary on processors 4-wide or wider
    - Core i7: is one load & one store per cycle
  - More important to support multiple loads than multiple stores
    - Insn mix: loads (~20–25%), stores (~10–15%)
  - Alpha 21164: two loads *or* one store per cycle

D$ Bandwidth

- 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
  - – Lots of wires; expensive in terms of latency, area (cost), and power

- Option#2: **banking** (or **interleaving**)
  - Divide D$ into "banks" (by address), one access per bank per 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

---

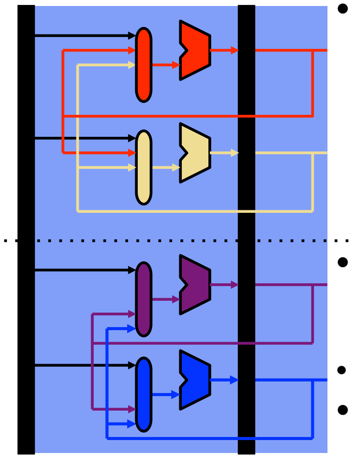
Not All N^2 Created Equal

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

- N^2 bypass … by far
  - 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)

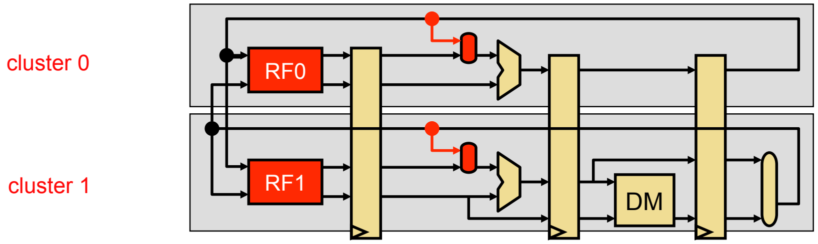
- Dependence cross-check not even 2nd biggest N^2 problem
  - Regfile is also an N^2 problem (think latency where N is #ports)
  - And also more serious than cross-check

Mitigating N^2 Bypass: Clustering



- **Clustering**: mitigates N^2 bypass
  - Group ALUs into **K** clusters
  - Full bypassing within a cluster
  - Limited bypassing between clusters
    - **With 1 or 2 cycle delay**
  - (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
  - Statically (compiler) or dynamically
- Hurts IPC, allows wide issue at same clock
- E.g., Alpha 21264
  - Bypass wouldn't fit into clock cycle
  - 4-wide, 2 clusters

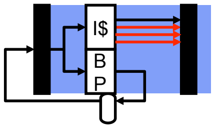
Mitigating N^2 RegFile: Clustering++



- **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
  - Alpha 21264: 4-way superscalar, two clusters

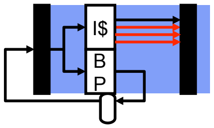
Superscalar "Front End"

Simple Superscalar Fetch



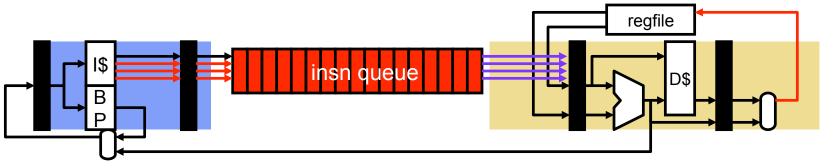
- What is involved in fetching multiple instructions per cycle?
- In same cache block? → no problem
  - 64-byte cache block is 16 instructions (~4 bytes per instruction)
  - Favors larger block size (independent of hit rate)
- What if next instruction is last instruction in a block?
  - Fetch only one instruction that cycle
  - Or, some processors may allow fetching from 2 consecutive blocks
- Compilers align code to I$ blocks (.align directive in asm)
  - Reduces I$ capacity
  - Increases fetch bandwidth utilization (more important)

Limits of Simple Superscalar Fetch



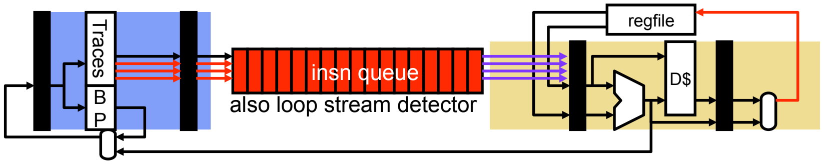
- How many instructions can be fetched on average?
  - BTB predicts the next block of instructions to fetch
    - Support multiple branch (direction) predictions per cycle
    - Discard post-branch insns after first branch predicted as "taken"
  - Lowers effective fetch width and IPC
  - Average number of instructions per taken branch?
    - Assume: 20% branches, 50% taken → ~10 instructions
- Consider a 5-instruction loop with an 4-issue processor
  - Without smarter fetch, ILP is limited to 2.5 (not 4)
- Compiler could "unroll" the loop (reduce taken branches)
- How else can we increase fetch rate?

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: predict next two blocks (extend BTB)
  - Transmits two PCs to fetch stage: "next PC" and "next-next PC"
  - Access I-cache twice (requires multiple ports or banks)
  - Requires extra merging logic to select and merge correct insns
  - – Elongates pipeline, increases branch penalty

Increasing Superscalar Fetch Rate



- Option #3: "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
- Option #4: trace cache (Pentium 4)
  - Tracks "traces" of disjoint but dynamically consecutive instructions
  - Pack (predicted) taken branch & its target into a one "trace" entry
  - Fetch entire "trace" while predicting the "next trace"

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 optmistic 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

Predication

- Branch mis-predictions hurt more on superscalar
  - Replace difficult branches with something else…
  - Convert control flow into data flow (& dependencies)
  - Helps hard-to-predict branches (but can hurt predictable branches)
- **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
    - – Doesn't handle conditional memory operations
    - + Only good way of adding predication to an existing ISA
- **If-conversion**: replacing control with predication

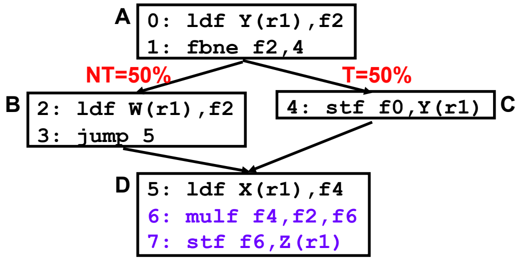
Predication If-Conversion Example

Source code

```
A = Y[i];
if (A == 0)
   A = W[i];
else
   Y[i] = 0;
Z[i] = A*X[i];
```

Machine code

```
0: ldf Y(r1),f2
1: fbne f2,4
2: ldf W(r1),f2
3: jump 5
4: stf f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```



Using Predication ↓

```
0: ldf Y(r1),f2
1: fspne f2,p1
2: ldf.p p1,W(r1),f2
4: stf.np p1,f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

ISA Support for Predication

```
0: ldf Y(r1),f2
1: fspne f2,p1
2: ldf.p p1,W(r1),f2
4: stf.np p1,f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

- Itanium: change branch 1 to **set-predicate insn `fspne`**
- Change insns 2 and 4 to **predicated insns**
  - `ldf.p` performs `ldf` if predicate `p1` is true
  - `stf.np` performs `stf` if predicate `p1` is false

CMOV Prediction Example

```
int func(int a, int b, int* array)
{
  if (a > 0) {
    return b;
  } else {
    return array[b];
  }
}
```

```
int func2(int a, int b, int* array)
{
  int temp = array[b];
  if (a > 0) {
    return b;
  } else {
    return temp;
  }
}
```

```
func:  testl   %edi, %edi
       jg      .L2
       movslq  %esi,%rax
       movl    (%rdx,%rax,4), %esi
.L2:   movl    %esi, %eax
       ret
```

```
func2: movslq  %esi, %rax
       testl   %edi, %edi
       cmovle  (%rdx,%rax,4), %esi
       movl    %esi, %eax
       ret
```

- x86 only has a "CMOV" instruction
  - Note: in x86's CMOV, any "load" part is non-conditional
- Small change in the code helps the compiler optimize

Another CMOV Example (Part I)

- gcc –Os –fno-if-conversion

```
tree_t* search(tree_t* t, int key)  L3:
{                                          cmpl   %esi, (%rdi)
  while (t != NULL) {                      je     L4
    if (t->value == key) {                 jle    L6
      return t;                            movq   8(%rdi), %rdi
    }                                      jmp    L12
                                    L6:
    if (t->value > key) {                  movq   16(%rdi), %rdi
      t = t->right_ptr;             L12:
    } else {                               testq  %rdi, %rdi
      t = t->left_ptr;                     jne    L3
    }
  }
  return NULL;
}
```

- Baseline
  - Same with and without –fno-in-conversion flag!

Another CMOV Example (Part II)

- gcc –Os –fno-if-conversion

```
tree_t* search(tree_t* t, int key)  L3:
{                                          cmpl   %esi, (%rdi)
  while (t != NULL) {                      je     L4
    if (t->value == key) {                 movq   8(%rdi), %rax
      return t;                            movq   16(%rdi), %rdi
    }                                      jle    L12
    tree_t* right = t->right_ptr;          movq   %rax, %rdi
    tree_t* left = t->left_ptr;     L12:
    if (t->value > key) {                  testq  %rdi, %rdi
      t = right;                           jne    L3
    } else {
      t = left;
    }
  }
  return NULL;
}
```

- Similar assembly as before (-fno-if-converstion)
  - Does reduce taken branches

Another CMOV Example (Part III)

- gcc –Os

```
tree_t* search(tree_t* t, int key)  L3:
{                                          cmpl   %esi, (%rdi)
  while (t != NULL) {                      je     L4
    if (t->value == key) {                 movq   16(%rdi), %rax
      return t;                            movq   8(%rdi), %rdi
    }                                      cmovle %rax, %rdi
    tree_t* right = t->right_ptr;   L22:
    tree_t* left = t->left_ptr;            testq  %rdi, %rdi
    if (t->value > key) {                  jne    L3
      t = right;
    } else {
      t = left;
    }
  }
  return NULL;
}
```

- Now, with –fif-converstion (enabled by default)
  - Uses CMOV to avoid branch misprediction

Multiple Issue Implementations

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 dumb and low power
  - E.g., TransMeta Crusoe (4-wide)
  - **Variant: Explicitly Parallel Instruction Computing (EPIC)**
    - A compromise: compiler does some, hardware does the rest
    - E.g., Intel Itanium (6-wide)
- **Dynamically-scheduled superscalar**
  - **Hardware extracts more ILP by on-the-fly reordering**
  - Core 2, Core i7 (4-wide), Alpha 21264 (4-wide)

Very Long Instruction Word (VLIW)

- Hardware-centric multiple issue problems
  - – Wide fetch/branch prediction, N^2 bypass, N^2 dependence checks
  - – Hardware solutions have been proposed: clustering, etc.

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

VLIW Advantages

- \+ Simpler instruction fetch
  - Fetch a bundle per cycle
- \+ Simpler dependence check logic
  - Compiler guarantees all instructions in bundle independent
- \+ Simpler branch prediction
  - Restrict to one branch per bundle
- By default, doesn't help bypasses or register file problems
  - **Which are the much bigger problems!**
  - Although clustering and replication can help VLIW, too
- Compiler-visible clustering possible in VLIW
  - Each "lane" of VLIW has "local" registers (read/written by this lane)
  - A few "global" registers (read/written by any lane) are used to communicate between lanes

VLIW Disadvantages

- – Code density
  - Lots of "no-ops" in bundles
- – Not compatible across machines of different widths
  - "not compatible" could mean programs would execute incorrectly
  - Or, "not compatible" can mean programs would execute slowly
  - Is non-compatibility worth all of this?
  - How did TransMeta deal with compatibility problem?
    - Dynamically translates x86 to internal VLIW
  - GPUs also use VLIW, do dynamic translation of graphics operations
- Finally, VLIW doesn't solve all problems
  - VLIW mainly targets dependence checking
    - Which isn't the worst N^2 problem in multiple-issue
  - Doesn't magical create ILP

EPIC

- **EPIC (Explicitly Parallel Insn Computing)**
  - Variant of VLIW (Variable Length Insn Words)
  - Implemented as "bundles" with explicit dependence bits
    - Helps code density
    - Code is compatible with different "bundle" width machines
  - E.g., **Intel Itanium** (IA-64)
    - 128-bit bundles (three 41-bit insns + 4 dependence bits)
  - **Still does not address bypassing or register file issues**

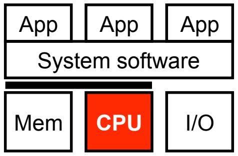
Multiple Issue Redux

- Multiple issue
  - Exploits insn level parallelism (ILP) beyond pipelining
  - Improves IPC, but perhaps at some clock & energy penalty
  - 4-6 way issue is about the peak issue width currently justifiable

- Problem spots
  - N^2 bypass & register file → clustering
  - Fetch + branch prediction → buffering, loop streaming, trace cache
  - N^2 dependency check → VLIW/EPIC (but unclear how key this is)

- Implementations
  - (Statically-scheduled) superscalar, VLIW/EPIC

Multiple Issue Summary



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

- Multiple-issue designs
  - "Superscalar"
  - VLIW