Remainder of CIS501: 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
  - After that: dynamic scheduling
    - Extract much more ILP via out-of-order processing
- Data-level parallelism (DLP)
  - Single-instruction, multiple data
  - Example: one instruction, four 16-bit adds (using 64-bit registers)
- Thread-level parallelism (TLP)
  - Multiple software threads running on multiple cores

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)

Readings

- Textbook (MA:FSPTCM)
  - Sections 3.1, 3.2 (but not “Sidebar” in 3.2), 3.5.1
  - Sections 4.2, 4.3, 5.3.3
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 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, 2 cycle penalty, no branch prediction

- Scalar pipeline
  - $1 + 0.2 \times 0.75 \times 2 = 1.3 \rightarrow 1.3/1 = 1.3 \rightarrow 30\%$ slowdown

- 2-way superscalar pipeline
  - $0.5 + 0.2 \times 0.75 \times 2 = 0.8 \rightarrow 0.8/0.5 = 1.6 \rightarrow 60\%$ slowdown

- 4-way superscalar
  - $0.25 + 0.2 \times 0.75 \times 2 = 0.55 \rightarrow 0.55/0.25 = 2.2 \rightarrow 120\%$ slowdown

**Simplest Superscalar: Split Floating Point**

- Split integer and floating point
- 1 integer + 1 FP
  - Limited modifications
    - Limited speedup
- Floating points loads in "integer" pipe

**A Four-issue Pipeline (2 integer, 2 FP)**

- 2 integer + 2 FP
- Similar to Alpha 21164

**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 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 $\rightarrow$ 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 $(N^2 \times P)$ 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 $\rightarrow$ 4 write ports

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

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?
    - $\sim4$ instruction per cycle maximum
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?
  - 2*N register read ports (latency \( \propto \) #ports)
- What about the **stall logic**?

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 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

\( N^2 \) Dependence Cross-Check

- Stall logic for 1-wide pipeline with full bypassing
  - Full bypassing \( \rightarrow \) load/use stalls only
    \( X/M,op==LOAD && (D/X.rs1==X/M.rd) \ OR \ D/X.rs2==X/M.rd) \)
  - Two "terms": \( \propto 2N \)
- Now: same logic for a 2-wide pipeline
  - \( X/M_{1}.op==LOAD && (D/X_{1}.rs1==X/M_{1}.rd) \ OR \ D/X_{1}.rs2==X/M_{1}.rd) \)
  - \( X/M_{2}.op==LOAD && (D/X_{2}.rs1==X/M_{2}.rd) \ OR \ D/X_{2}.rs2==X/M_{2}.rd) \)
  - Eight "terms": \( \propto 2N^2 \)
    - **\( N^2 \) dependence cross-check**
  - Not quite done, also need
    - \( D/X_{1}.rs1==D/X_{1}.rd \) OR \( D/X_{1}.rs2==D/X_{1}.rd \)
Dollar 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
  - Lots of wires; 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, little latency penalty
  - Not a solution for stores (that’s OK), area (cost), power penalty

Dollar Bandwidth: Banking

- Option #3: 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

- Which address bit(s) determine bank?
  - Offset bits? Individual cache lines spread among 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

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 ∝ #ports²
  - In reality, fewer than that
    - Read ports: some instructions read only one register
    - Write ports: stores, branches (35% insns) don’t write registers
  - Multi-porting and replication both work for reg. files
  - Alpha 21264 uses replication (more later)
  - Banking? Not really a good option

Superscalar Bypass

- \(N^2\) bypass network
  - \(N+1\) input muxes at each ALU input
  - \(N^2\) point-to-point connections
  - Routing lengthens wires (slow)
  - 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
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 stores 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

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 \(\Rightarrow\) \(\sim10\) 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"

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)**
  - Compiler identifies independent instructions, requires new binaries
  + Hardware can be dumb and low power
  - 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**
  - Core 2, Core i7 (4-wide), Alpha 21264 (4-wide)
  - We've already talked about statically-scheduled superscalar
**VLIW**

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

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

**What Does VLIW Actually Buy You?**

+ Simpler I$/branch prediction
  - Relies on compiler and predication to avoid taken branches
+ Simpler dependence check logic
  - Compiler guarantees all instructions in bundle independent
- Doesn’t help bypasses or register file
  - 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**

- **EPIC (Explicitly ParallelInsn Computing)**
  - New VLIW (Variable LengthInsn 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 insns + 4 dependence bits)
  - **Still does not address bypassing or register file issues**

**Trends in Single-Processor Multiple Issue**

<table>
<thead>
<tr>
<th>Year</th>
<th>486</th>
<th>Pentium</th>
<th>PentiumII</th>
<th>Pentium4</th>
<th>Itanium</th>
<th>ItaniumII</th>
<th>Core2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

- 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
    - Out-of-order execution (or VLIW/EPIC)

- For high-performance **per watt** cores, issue width is \( \sim 2 \)
  - Advanced scheduling techniques not needed
  - Multi-threading (a little later) helps cope with cache misses
Multiple Issue Redux

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

- Problem spots
  - \( N^2 \) bypass \( \rightarrow \) clustering
  - Register file \( \rightarrow \) replication
  - Fetch + branch prediction \( \rightarrow \) buffering, loop streaming, trace cache

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

- Are there more radical ways to address these challenges?

Grid Processor

- Components
  - next block logic/predictor (NH), I\$, D\$, regfile
  - NxN ALU grid: here 4x4

- Pipeline stages
  - Fetch block to grid
  - Read registers
  - Execute/memory
    - Cascade
  - Write registers

- Block atomic
  - No intermediate regs
  - Grid limits size/shape

Research: Grid Processor

- **Grid processor** (TRIPS) [Nagarajan+, MICRO’01]
  - EDGE (Explicit Dataflow Graph Execution) execution model
  - Holistic attack on many fundamental superscalar problems
    - Specifically, the nastiest one: \( N^2 \) bypassing
    - But also \( N^2 \) dependence check
    - And wide-fetch + branch prediction
  - **Two-dimensional VLIW**
    - Horizontal dimension is insns in one parallel group
    - Vertical dimension is several vertical groups
    - Executes atomic code blocks
      - Uses predication and special scheduling to avoid taken branches

- UT-Austin research project
  - Fabricated an actual chip with help from IBM

Aside: SAXPY

- **SAXPY** (Single-precision A X Plus Y)
  - Linear algebra routine (used in solving systems of equations)
  - Part of early “Livermore Loops” benchmark suite

```
for (i=0;i<N;i++)
Z[i]=A*X[i]+Y[i];
```

0: ldf X(r1),f1 // loop
1: mul f0,f1,f2 // A in f0
2: ldf Y(r1),f3 // X,Y,Z are constant addresses
3: add f2,f3,f4
4: stf f4,Z(r1)
5: addi r1,4,r1 // i in r1
6: blt r1,r2,0 // N*4 in r2
Grid Processor SAXPY

- A code block for this Grid processor has five 4-inxn words
  - Atomic unit of execution
- Some notes about Grid ISA
  - `read`: read register from register file
  - `pass`: null operation
  - `-1, 0, 1`: routing directives send result to next word
    - one insn left (-1), insn straight down (0), one insn right (1)
    - Directives specify value flow, no need for interior registers

Grid Processor SAXPY Cycle 1

- Map code block to grid

Grid Processor SAXPY Cycle 2

- Read registers

Grid Processor SAXPY Cycle 3

- Execute first grid row
- Execution proceeds in “data flow” fashion
  - Not lock step
Grid Processor SAXPY Cycle 4

- Execute second grid row

Grid Processor SAXPY Cycle 5

- Execute third grid row
  - Recall, \texttt{mulf} takes 5 cycles

Grid Processor SAXPY Cycle 6

- Execute third grid row

Grid Processor SAXPY

- When all instructions are done
  - Write registers and next code block PC
Grid Processor SAXPY Performance

- Performance
  - 1 cycle fetch
  - 1 cycle read regs
  - 8 cycles execute
  - 1 cycle write regs
  - 11 cycles total

- Utilization
  - $7 / (11 \times 16) = 4\%$

- What’s the point?
  + Simpler components
  + Faster clock?

Grid Processor Redux

+ No hardware dependence checks ... period
  - Insn placement encodes dependences, still get dynamic issue

+ Simple, forward only, short-wire bypassing
  - No wraparound routing, no metal layer crossings, low input muxes

- Code size
  - Lots of \texttt{nop} and \texttt{pass} operations

- Non-compatibility
  - Code assumes horizontal \textit{and} vertical grid layout

- No scheduling between hyperblocks
  - Can be overcome, but is pretty nasty

- Poor utilization
  - Overcome by multiple concurrent executing hyperblocks

- Interesting: TRIPS has morphed into something else
  - Grid is gone, replaced by simpler “window scheduler”
  - Forward nature of ISA exploited to create tiled implementations

Next Up...

- Extracting more ILP via dynamic scheduling in hardware