This Unit: (In-Order) Superscalar Pipelines

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

- Multiple-issue designs
  - “Superscalar”
  - VLIW

Readings

- P+H
  - Chapter 6.9

Scalar Pipelines

- So far we have looked at scalar pipelines
  - One instruction per stage
  - With control speculation
  - With bypassing (not shown)
  - With floating-point ...
Floating Point Pipelines

- Floating point (FP) insns typically use separate pipeline
  - Splits at decode stage: at fetch you don't know it's a FP insn
  - Most (all?) FP insns are multi-cycle (here: 3-cycle FP adder)
  - Separate FP register file
  - FP loads and stores execute on integer pipeline (address is integer)

The “Flynn Bottleneck”

- Performance limit of scalar pipeline is CPI = IPC = 1
  - Hazards → limit is not even achieved
  - Hazards + latch overhead → diminishing returns on “super-pipelining”

The “Flynn Bottleneck”

- Overcome IPC limit with **super-scalar pipeline**
  - Two insns per stage, or three, or four, or six, or eight...
  - Also called **multiple issue**
  - Exploit **Instruction-Level Parallelism (ILP)**

Superscalar Pipeline Diagrams - Ideal

<table>
<thead>
<tr>
<th>scalar</th>
<th>1 2 3 4 5 6 7 8 9 10 11 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw 0(r1)</td>
<td>r2</td>
</tr>
<tr>
<td>lw 4(r1)</td>
<td>r3</td>
</tr>
<tr>
<td>lw 8(r1)</td>
<td>r4</td>
</tr>
<tr>
<td>add r14, r15</td>
<td>r6</td>
</tr>
<tr>
<td>add r12, r13</td>
<td>r7</td>
</tr>
<tr>
<td>add r17, r16</td>
<td>r8</td>
</tr>
<tr>
<td>lw 0(r18)</td>
<td>r9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2-way superscalar</th>
<th>1 2 3 4 5 6 7 8 9 10 11 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw 0(r1)</td>
<td>r2</td>
</tr>
<tr>
<td>lw 4(r1)</td>
<td>r3</td>
</tr>
<tr>
<td>lw 8(r1)</td>
<td>r4</td>
</tr>
<tr>
<td>add r14, r15</td>
<td>r6</td>
</tr>
<tr>
<td>add r12, r13</td>
<td>r7</td>
</tr>
<tr>
<td>add r17, r16</td>
<td>r8</td>
</tr>
<tr>
<td>lw 0(r18)</td>
<td>r9</td>
</tr>
</tbody>
</table>
Superscalar Pipeline Diagrams - Realistic

### scalar
- `lw 0(r1) r2`
- `lw 4(r1) r3`
- `lw 8(r1) r4`
- `add r4, r5 r6`
- `add r2, r3 r7`
- `add r7, r6 r8`
- `lw 0(r8) r9`

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D*</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D*</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>d*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D*</td>
<td>X</td>
</tr>
</tbody>
</table>

### 2-way superscalar
- `lw 0(r1) r2`
- `lw 4(r1) r3`
- `lw 8(r1) r4`
- `add r4, r5 r6`
- `add r2, r3 r7`
- `add r7, r6 r8`
- `lw 0(r8) r9`

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D*</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>F</td>
<td>d*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D*</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>d*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D*</td>
<td>X</td>
</tr>
</tbody>
</table>

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**
    - $1 + 0.2 \times 0.75 \times 2 = 1.3 \rightarrow 1.3/1 = 1.3 \rightarrow 30\% slowdown$
  - **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**
    - $0.25 + 0.2\times 0.75\times 2 = 0.55 \rightarrow 0.55/0.25 = 2.2 \rightarrow 120\% slowdown$

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

Challenges for Superscalar Pipelines

- **So you want to build an N-way superscalar...**
- **Hardware challenges**
  - Stall logic: $N^2$ terms
  - Bypasses: $2N^2$ paths
  - Register file: $3N$ ports
  - IMem/DMem: how many ports?
  - Anything else?
- **Software challenges**
  - Does program inherently have ILP of $N$?
  - Even if it does, compiler must schedule code to expose it
- **Given these challenges, what is a reasonable $N$?**
  - Current answer is 3 or 4
Superscalar "Execution"

- N-way superscalar = N of every kind of functional unit?
  - N ALUs? OK, ALUs are small and integer insns are common
  - N FP dividers? No, FP dividers are huge and fdiv is uncommon
  - How many loads/stores per cycle? How many branches?

Superscalar Register File

- Except DMem, execution units are easy
  - Getting values to/from them is the problem

  - 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 \( \propto \#\text{ports}^2 \propto (3N)^2 \) (slow for large N)

Superscalar Execution

- Common design: functional unit mix \( \propto \) insn type mix
  - Integer apps: 20–30% loads, 10–15% stores, 15–20% branches
  - FP 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 (including 2 loads or 1 store) + 2 FP

- 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)

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
Superscalar Stall Logic

- Full bypassing → load/use stalls only
  - Ignore 2nd register input

- Stall logic for scalar pipeline
  \[(X/M, op==LOAD && D/X.rs1==X/M.rd)\]

- Stall logic for a 2-way superscalar pipeline
  \[(X/M_1, op==LOAD && D/X_1.rs1==X/M_1.rd) || (X/M_2, op==LOAD && D/X_2.rs1==X/M_2.rd)\]

- Stall logic for younger insn in pair: doesn’t stall older insn
  \[(X/M_1, op==LOAD && D/X_2.rs1==X/M_1.rd) || (X/M_2, op==LOAD && D/X_2.rs1==X/M_2.rd)\]

- 5 terms for 2 insns: \(N^2\) dependence cross-check
  - Actually \(N^1+N-1\)

Not All \(N^2\) Problems Created Equal

- \(N^2\) bypass vs. \(N^2\) 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

Superscalar Pipeline Stalls

- If older insn in pair stalls, younger insns must stall too
- What if younger insn stalls?
  - Can older insn from next group move up?
    - **Fluid**: yes
      + Helps CPI a little, hurts clock a little
    - **Rigid**: no
      + Hurts CPI a little, but doesn’t impact clock

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 data bus
  - Most tricky aspects involve branch prediction

- What about Decode?
  - Easier with fixed-width instructions (MIPS, Alpha, PowerPC, ARM)
  - Harder with variable-length instructions (x86)
    - Can be pipelined

Superscalar Fetch with Branches

- Three related questions
  - How many branches are predicted per cycle?
  - If multiple insns fetched, which is assumed to be the branch?
  - Can we fetch across the branch if it is predicted “taken”?

- Simplest, common design: “one”, “doesn’t matter”, “no”
  - One prediction, discard post-branch insns if prediction is “taken”
  - Doesn’t matter: associate prediction with non-branch to same effect
    - Lowers effective fetch bandwidth width and IPC
  - Average number of insns per taken branch? \( \sim 8–10 \) in integer code

- Compiler can help
  - Reduce taken branch frequency: e.g., unroll loops

Pipelined Branch Prediction / Fetch

- To fetch across a taken branch...
  - Must fetch from two separate IMem addresses in same cycle
    - Split IMem into even/odd “banks” to provide bandwidth for this
  - Pipeline branch prediction and fetch: branch prediction first
  - Branch prediction sends two PCs to fetch: PC & target PC (if any)
    - Elongates pipeline, increases branch penalty
  - Pentium II & III do something like this

Aside: VLIW

- **VLIW: Very Long Insn Word**
  - 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 \texttt{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)
    - Can construct appearance of full predication from one primitive
      \[ \text{cmov}_{eq} r1, r2, r3 \quad // \text{if} \ (r1==0) \ r3=r2; \]
    - 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