Instruction Level Parallelism (ILP)

- ILP is the overlapping of instruction execution.
  - pipelining
  - multiple issue
- scheduling approaches to exploit ILP
  - static (compiler)
  - dynamic (hardware)

Data Dependences

- Instructions must be independent to be executed in parallel.
- types of data dependences
  - True dependences can lead to RAW hazards.
  - name dependences
    - Antidependences can lead to WAR hazards.
  - Output dependences can lead to WAW hazards.

Control Dependences

- An instruction is control dependent on a branch instruction if the instruction will only be executed when the branch has a specific result.
- control dependent restrictions
  - An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
  - An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.
If a hazard is detected that cannot be resolved, then the instruction is stalled in the ID stage and no new instructions are fetched or issued until the dependence is cleared.

Compiler techniques are used to statically schedule the instructions to avoid or minimize stalls.

- Increase the distance between dependent instructions.
- Perform other transformations (e.g., register renaming).

Instruction Scheduling

```
loop: L.D F0,O(R1)  # original assembly code
    ADD.D F4,F0,F2
    S.D F4,O(R1)
    DADDUI R1,R1,#-8
    BNE R1,R2,loop

loop: L.D F0,O(R1)  # original code with stalls
   stall
    ADD.D F4,F0,F2
   stall
    S.D F4,O(R1)
    DADDUI R1,R1,#-8
   stall
    BNE R1,R2,loop

loop: L.D F0,O(R1)  # scheduled code
    DADDUI R1,R1,#-8
    ADD.D F4,F0,F2
    S.D F4,8(R1)
    BNE R1,R2,loop
```

Loop Unrolling

```
for (i = 0; i < n; i++)
    a[i] = a[i] + x;
=>
for (i = 0; i < n/4; i++)
    a[i] = a[i] + x;
for (; i < n; i++) {
    a[i] = a[i] + x; i++;
    a[i] = a[i] + x; i++;
    a[i] = a[i] + x; i++;
    a[i] = a[i] + x;
}
=>
for (i = 0; i < n/4; i++)
    a[i] = a[i] + x;
for (; i < n; i += 4) {
    a[i] = a[i] + x;
    a[i+1] = a[i+1] + x;
    a[i+2] = a[i+2] + x;
    a[i+3] = a[i+3] + x;
}
```

Loop Unrolling at the Assembly Level

```
unrolled loop          after moving DADDUIs

loop:                loop:
  L.D F0,O(R1)       L.D F0,O(R1)
  ADD.D F4,F0,F2     ADD.D F4,F0,F2
  S.D F4,O(R1)       S.D F4,O(R1)
  DADDUI R1,R1,#-8   DADDUI R1,R1,#-8
  L.D F6,O(R1)       ADD.D F6,F6,F2
  ADD.D F6,F6,F2     S.D F8,F8,F2
  S.D F8,O(R1)       ADD.D F8,F8,F2
  DADDUI R1,R1,#-8   ADD.D F10,-16(R1)
  L.D F10,O(R1)      ADD.D F12,F10,F2
  ADD.D F12,F10,F2    S.D F12,F12(-16(R1)
  S.D F12,O(R1)      ADD.D F16,F14,F2
  DADDUI R1,R1,#-8   DADDUI R1,R1,#-8
  L.D F14,O(R1)      DADDUI R1,R1,#-8
  ADD.D F14,F14,F2    DADDUI R1,R1,#-8
  S.D F16,O(R1)      DADDUI R1,R1,#-8
  DADDUI R1,R1,#-8   DADDUI R1,R1,#-8
  BNE R1,R2,loop     BNE R1,R2,loop
```
Loop Unrolling and Scheduling

<table>
<thead>
<tr>
<th>Action</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>after coalesing adds</td>
<td>L.D F0,0(R1)</td>
</tr>
<tr>
<td>after showing stalls</td>
<td>L.D F0,0(R1)</td>
</tr>
<tr>
<td></td>
<td>ADD.D F12,F10,F2</td>
</tr>
<tr>
<td></td>
<td>S.D F6,-16(R1)</td>
</tr>
<tr>
<td></td>
<td>S.D F8,-8(R1)</td>
</tr>
<tr>
<td></td>
<td>L.D F10,-16(R1)</td>
</tr>
<tr>
<td></td>
<td>ADD.D F8,F6,F2</td>
</tr>
<tr>
<td></td>
<td>S.D F8,-8(R1)</td>
</tr>
<tr>
<td></td>
<td>L.D F10,-16(R1)</td>
</tr>
<tr>
<td></td>
<td>ADD.D F12,F10,F2</td>
</tr>
<tr>
<td></td>
<td>S.D F12,-16(R1)</td>
</tr>
<tr>
<td></td>
<td>ADD.D F16,F14,F2</td>
</tr>
<tr>
<td></td>
<td>DADDUI R1,R1,#-32</td>
</tr>
<tr>
<td></td>
<td>S.D F8,-8(R1)</td>
</tr>
<tr>
<td></td>
<td>L.D F14,-24(R1)</td>
</tr>
<tr>
<td></td>
<td>L.D F14,-24(R1)</td>
</tr>
<tr>
<td></td>
<td>S.D F16,-24(R1)</td>
</tr>
<tr>
<td></td>
<td>S.D F16,-24(R1)</td>
</tr>
<tr>
<td></td>
<td>DADDUI R1,R1,#-32</td>
</tr>
<tr>
<td></td>
<td>DADDUI R1,R1,#-32</td>
</tr>
<tr>
<td></td>
<td>L.D F16,-24(R1)</td>
</tr>
<tr>
<td></td>
<td>S.D F16,-24(R1)</td>
</tr>
<tr>
<td></td>
<td>L.D F16,-24(R1)</td>
</tr>
<tr>
<td></td>
<td>DADDUI R1,R1,#-32</td>
</tr>
<tr>
<td></td>
<td>S.D F16,-24(R1)</td>
</tr>
<tr>
<td></td>
<td>BNE R1,R2,loop</td>
</tr>
<tr>
<td></td>
<td>BNE R1,R2,loop</td>
</tr>
<tr>
<td></td>
<td>BNE R1,R2,loop</td>
</tr>
</tbody>
</table>

Dependences and Instruction Scheduling

- Dependences restrict the movement of instructions and the order in which results must be calculated.
- **Data (true) dependences**
  - True dependences cannot be avoided.
- **Name dependences**
  - Anti and output dependences can sometimes be avoided by renaming.
- **Control dependences**
  - Control dependences can sometimes be avoided through branch elimination techniques or speculation.

True Dependences

- Data (true) dependences occur when the result of one instruction is used by another and can lead to RAW hazards.
- The following sequence of instructions cannot be reordered.

<table>
<thead>
<tr>
<th>Action</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L.D F0,0(R1)</td>
</tr>
<tr>
<td></td>
<td>ADD.D F4,F0,F2</td>
</tr>
<tr>
<td></td>
<td>S.D F4,O(R1)</td>
</tr>
</tbody>
</table>
Can sometimes be avoided by using additional registers (register renaming).

### Correlating Branch Prediction Buffers

- An \((m,n)\) predictor uses the behavior of the last \(m\) branches executed to choose from \(2^m\) branch predictors, each of which is \(n\)-bits.
- A \((0,1)\) predictor would be the conventional 1-bit predictor.
- A \((0,2)\) predictor would be the conventional 2-bit predictor. Usually machines never have more than 2 bits for each predictor.
- A correlating predictor records the most recent \(m\) branch results using an \(m\)-bit shift register, where each bit indicates if the corresponding branch was taken or not taken.
- Consider the following example with a \((2,1)\) predictor.

```c
for (i = 0; i < 100; i++)
    if (i & 1) /* i is odd */
        ...
```

### Correlating Branch Predictor Example

Below is an example of a \((2,1)\) branch-prediction buffer. It uses a 2-bit global history to choose from among four 1-bit predictors for each branch address with a total of 32 entries.
Tournament Branch Prediction Buffers

- Uses both a local and a global predictor.
- Uses a selector, similar to the 2-bit predictor, to decide whether to use the result of the local or global predictor.
- Are the most accurate types of branch predictors in use today.

![Branch Prediction Diagram]

Branch Target Buffer

- Stores the actual branch target address.
- Checked during the IF stage to allow earlier resolution.
- A tag is needed to make sure the instruction is a branch.
- Not put into the buffer until the branch is taken.

![Branch Target Buffer Diagram]

Branch Folding

- Can store with the target address in the BTB one or more instructions at the branch target.
- The branch target instruction(s) can be delivered to the ID stage in the following cycle.
- Need to adjust the target address in BTB to be after the target instruction(s).
- Allows for zero-cycle unconditional jumps and sometimes zero-cycle taken branches.
- Has the effect of folding the branch in with the target instruction(s).

![Branch Folding Diagram]

Return Prediction

- Used for predicting returns.
- Uses a circular buffer of return addresses called the return address stack (RAS).
- Pushes the return address on stack at each call.
- Pops the return address at each return.
- Can either add a bit to the BTB or keep a predecode bit in the L1 IC to recognize that the current instruction is a return during the instruction fetch stage so the address on the top of the RAS should be used.
- Works well as long as the calling depth does not exceed the number of entries in the stack, a context switch does not occur, and a callee always returns to the caller (no longjmp operations).
Dynamic Scheduling

- The hardware is designed to rearrange the order in which instructions are executed to reduce pipeline stalls.
- Advantages include avoiding more stalls and simplifying the compiler. Can be used to avoid WAW and WAR hazards.
- Disadvantages include more complicated hardware, more energy usage, and perhaps a somewhat longer cycle time.

Out-of-Order Execution

- Dynamic scheduling allows out-of-order (OoO) instruction execution.
- In the example below, the SUB.D can execute while the ADD.D is stalled as the SUB.D does not depend on the DIV.D or the ADD.D.

```
DIV.D F0,F2,F4
ADD.D F10,F0,F8
SUB.D F12,F8,F14
```

- The ID stage is split into two parts.
  - Issue: Decode instructions and check for structural hazards.
  - Read Operands: Wait until no RAW hazards.
- OoO execution can result in OoO completion, which complicates exception handling.

Scoreboarding Approach

- Was used to allow OoO execution when there are no structural hazards and no data dependences.
- Was used in the CDC 6600 scoreboard.
- Can be achieved by using multiple function units.

Scoreboarding Steps

- Issue - Get an instruction from the instruction queue. Issue it if the scoreboard indicates that there is an available functional unit and no other active instruction has the same destination register. Otherwise stall the issue stage. Checks for structural and WAW hazards.
- Read operands - When the source operands for an active instruction in a function unit are available, read the operands from the register file and indicate to the scoreboard that execution may start. Checks for RAW hazards.
- Execute - Start execution when operands have been read. Notify the scoreboard when execution is completed.
- Write result - If there is a preceding active instruction that has not read one of its operands that is the same register as the destination of the completing instruction, then stall the completing instruction. Otherwise store the result to the register file. Checks for WAR hazards.
Tomasulo Approach

- Is used to avoid WAR and WAW hazards.
- Data values are often directly obtained from buffers or functional units via a common data bus (CDB).
- Uses dynamic register renaming, which is accomplished by reservation stations. As instructions are issued, the register specifiers for pending operands are renamed to be the names of the reservation stations or load buffer entries.
- Uses dynamic scheduling (out-of-order execution).
- Uses dynamic memory disambiguation.

Tomasulo Algorithm Steps

- Issue - Get an instruction from the instruction queue. Issue it if there is an empty reservation station and send the operands to this reservation station if they are available in registers. Checks for structural hazards.
- Execute - Monitor the CDB waiting for the operands to become available. When both operands are available, execute the operation. Checks for RAW hazards.
- Write result - Write the result on the CDB when it is available. Indicate which register will receive that value. The value will also be placed in the appropriate reservation stations waiting for the result.

Tomasulo Algorithm Example 1

- execution assumptions
  - one cycle for integer ALU operations
  - two cycles for a load (one for address calculation and one for data cache access)
  - two cycles for ADD.D, 4 cycles for MUL.D, 10 cycles for DIV.D

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issues at</th>
<th>Execute Start</th>
<th>Execute End</th>
<th>Memory Access at</th>
<th>Write CDB at</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F6,34(R2)</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>L.D F2,45(R3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL.D F0,F2,F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB.D F8,F6,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV.D F10,F0,F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D F6,F8,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tomasulo Algorithm Example 2

- execution assumptions
  - one cycle for integer ALU operations and branches
  - two cycles for a load and stores
  - four cycles for MUL.D

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issues at</th>
<th>Execute Start</th>
<th>Execute End</th>
<th>Memory Access at</th>
<th>Write CDB at</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>MUL.D F4,F0,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADDIU R1,R1,#-8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE R1,R2,Loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F0,0(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL.D F4,F0,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADDIU R1,R1,#-8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE R1,R2,Loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Hardware-Based Speculation**

- Combines three key ideas.
  - Dynamic branch prediction
  - Speculation of instructions before control dependences are resolved
  - Dynamic scheduling (out-of-order execution)
- We will see that instructions on such a machine:
  - Issue in order.
  - Can execute out of order.
  - Update the state of the machine (commit) in order.

**Speculative Tomasulo Approach**

- Allows speculative execution with dynamic scheduling based on Tomasulo’s algorithm.
- Like Tomasulo’s algorithm, it allows instructions to execute out of order.
- Unlike Tomasulo’s algorithm, it forces the instructions to commit (update registers or memory) in order. The advantages are:
  - Supports speculative execution.
  - Precise exceptions are supported.

**Speculative Tomasulo Steps**

- Issue - Get an instruction from the instruction queue. Issue it if there is an empty reservation station and an empty slot in the reorder buffer and mark both as busy. Checks for structural hazards.
- Execute - Monitor the CDB waiting for the operands to become available. When both operands are available, execute the operation. Checks for RAW hazards.
- Write result - Write the result on the CDB when it is available with the appropriate tag (reorder buffer slot #) that was stored in the reservation station. Mark the reservation station as available. Write result also into reorder buffer.
- Commit - When the result of the instruction at the head of the reorder buffer is available, update the state (memory or register) and remove the instruction from the reorder buffer. If a branch is at the head and it is found that it was predicted incorrectly, then flush the branch and the instructions that entered after the branch out of the buffer.

**Speculative Tomasulo Algorithm Example 1**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issues at</th>
<th>Executes at</th>
<th>Memory Access at</th>
<th>Write CDB at</th>
<th>Commits at</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F6,34(R2)</td>
<td>1</td>
<td>2-2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>L.D F2,45(R3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL.D F0,F2,F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB.D F8,F6,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV.D F10,F0,F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D F6,F8,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Speculative Tomasulo Algorithm Example 2

- **execution assumptions**
  - one cycle for integer ALU operations and branches
  - two cycles for a load and stores (one for address calculation and one for data cache access)
  - four cycles for MUL.D

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issues at</th>
<th>Executes at</th>
<th>Memory Access at</th>
<th>Write CDB at</th>
<th>Commits at</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>1</td>
<td>2-2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>MUL.D F4,F0,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADDIU R1,R1,#-8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE R1,R2,Loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F0,0(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL.D F4,F0,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADDIU R1,R1,#-8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE R1,R2,Loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic Scheduling Techniques Summary

- Scoreboarding introduced in the CDC 6600. Introduced out-of-order execution.
- Tomasulo algorithm introduced in the IBM 360/91. Supported register renaming to avoid WAR and WAW hazards.
- Speculative Tomasulo algorithm, which was used in the PowerPC 620, MIPS 10000, Intel Pentium Pro, etc. Supported speculation and precise exceptions.

Hardware versus Software Speculation

- **hardware speculation advantages**
  - better memory disambiguation
  - better branch prediction
  - simpler compiler
  - binary code compatibility

- **hardware speculation disadvantages**
  - more complex design
  - more space on chip
  - May slow the processor clock cycle.

How Much to Speculate?

- Speculation is not free due to branch mispredictions.
  - Often causes more energy usage due to unnecessarily executed instructions and the cost of recovery.
  - Can potentially result in a performance degradation, so only low-cost exceptional events (e.g. L1 DC miss) are allowed for speculative instructions.
  - Some additional complexity when speculating across multiple branches.
Value Prediction

- Value prediction attempts to predict the value that will be produced by an instruction.
- Value prediction is a form of speculation.
- Much of the research on value prediction has been on loads.
  - Determine which loads are predictable by using profile data and mark these loads.
  - Index into table using address of instruction to verify that previous load result for the instruction is in the table when the tag matches and speculate using last value loaded.
- Must be a method for recovery when the prediction is incorrect.

Issuing Multiple Instructions in Parallel

- Issuing multiple instructions in parallel requires simultaneous actions to be performed for multiple instructions.
  - fetching
  - decoding
  - executing
  - writing results
- multiple issue processors
  - statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - dynamically scheduled superscalar processors

Statically Scheduled Superscalar Approach

- Issue multiple instructions that can be quickly detected as independent.
- Typically, superscalar machines can issue at most 2 to 4 instructions simultaneously, with 2 instructions being most typical.
- If one instruction in the set has some dependency, then only the preceding instructions will be issued.
- More likely to have hazards causing stalls.
- No increase in code size and less work for the compiler.
- Programs compiled for nonsuperscalar machines will still execute and have benefits.

VLIW Approach

- Can issue many instructions in parallel.
- The compiler has the responsibility to package multiple independent instructions together.
- Simplifies issuing hardware.
- With many functional units, this requires complex compiler scheduling support.
  - loop unrolling
  - software pipelining
  - superblock scheduling
VLIW Challenges

- Performing multiple loads and stores in the same cycle.
- There could be multiple accesses to the register file in the same cycle.
- The clock cycle may have to be lengthened.
- May significantly increase code size due to inserting no op instructions and compilation techniques (e.g. loop unrolling) to extract more parallelism.
- Any functional unit stall will cause the entire processor to stall. This includes cache misses.
- Different implementations (number of functional units) results in executables that are not binary compatible.

Dynamically Scheduled Superscalar Approach

- Can fetch multiple instructions simultaneously.
- Can issue multiple instructions in the same cycle.
- Write multiple results on the CDB in the same cycle.
- Commit multiple instructions in the same cycle.

Power Is a Limiting Factor for Multiple-Issue Processors

- Techniques to boost performance of multiple-issue processors increase power consumption more than performance.
- Issuing more instructions in parallel requires logic overhead that grows faster than the issue rate.
- There is a growing gap between peak issue rates and sustained issue rates.
- Speculation is inherently inefficient since it can never be perfect.

Perfect Processor

- Infinite number of virtual registers available for register renaming. So all WAW and WAR hazards are avoided.
- Branch and jump prediction is perfect and infinite instruction prefetch buffer. So an unbounded buffer of instructions is available for execution.
- All memory addresses are known exactly. So loads can be moved up before stores whenever possible (or vice versa).
- No cache misses.
- Unlimited number of functional units.
- Only one cycle latencies for operations.
Limits Preventing a Perfect Processor

- The number of functional units must be limited.
- The number of comparisons to check for issue dependences in a window of instructions grows quadratically ($n^2 - n$). The available parallelism is reduced as the window size is decreased.
- Branch prediction levels can be high, but never perfect.
- Cannot have an infinite number of registers, but can do pretty well with a reasonable number.
- Perfect alias analysis is impossible at compile-time and very expensive at run-time.
- Cache misses and function unit latencies must also be taken into account.

Multithreading

- Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion.
- A processor must save the state of each process (register file, PC, stack pointer, condition codes) on a context switch.
- Thread switches must be more efficient than process switches.
- types of multithreading
  - Coarse-grained multithreading is the conventional multithreading and it switches threads on costly stalls (e.g. L2 misses).
  - Fine-grained multithreading switches between threads on each clock cycle in a round-robin fashion.
  - Simultaneous multithreading allows multiple threads to run simultaneously by exploiting the resources of a multiple-issue, dynamically scheduled processor.

SMT Design Challenges

- Dealing with a larger register file.
- Instruction issue is more complex, which could affect the clock cycle.
- Instruction commit is more complex.
- There may be more cache and TLB conflicts, which could degrade performance.

Fallacies and Pitfalls

- Fallacy: Processors with lower CPIs will always be faster.
- Fallacy: Processors with faster clock rates will always be faster.
- Pitfall: Sometimes bigger and dumber is better.