Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **Vector Processing**: Explicit coding of independent loops as operations on large vectors of numbers
  - Multimedia instructions being added to many processors
- **Superscalar**: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, Pentium III/4
- **(Very) Long Instruction Words (V)LIW**: fixed number of instructions (4-16) scheduled by the compiler; put ops into wide templates
  - Intel Architecture-64 (IA-64) 64-bit address
    - Renamed: "Explicitly Parallel Instruction Computer (EPIC)"
- Anticipated success of multiple instructions lead to Instructions Per Clock_cycle (IPC) vs. CPI

### Multiple Issue Issues

- **issue packet**: group of instructions from fetch unit that could potentially issue in 1 clock
  - If instruction causes structural hazard or a data hazard either due to earlier instruction in execution or to earlier instruction in issue packet, then instruction does not issue
  - 0 to N instruction issues per clock cycle, for N-issue
- **Performing issue checks in 1 cycle could limit clock cycle time**: $O(n^2-n)$ comparisons
  - => issue stage usually split and pipelined
- 1st stage decides how many instructions from within this packet can issue, 2nd stage examines hazards among selected instructions and those already been issued
  - => higher branch penalties => prediction accuracy important
Multiple Issue Challenges

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations AND No hazards
- If more instructions issue at same time, greater difficulty of decode and issue:
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue; (N-issue ~O(N²-N) comparisons)
  - Register file: need 2x reads and 1x writes/cycle
  - Rename logic: must be able to rename same register multiple times in one cycle! For instance, consider 4-way issue:

```
add r1, r2, r3       add p11, p4, p7
sub r4, r1, r2  =>  sub p22, p11, p4
lw  r5, 4(r4)      lw  p23, 4(p22)
add r5, r1, r2      add p12, p23, p4
```

Imagine doing this transformation in a single cycle!
- Result buses: Need to complete multiple instructions/cycle
  » So, need multiple buses with associated matching logic at every reservation station.
  » Or, need multiple forwarding paths

Dynamic Scheduling in Superscalar
The easy way

- How to issue two instructions and keep in-order instruction issue for Tomasulo?
  - Assume 1 integer + 1 floating point
  - 1 Tomasulo control for integer, 1 for floating point
- Key is assigning reservation station and updating control tables
  - Issue 2X Clock Rate, so that issue remains in order
- Only loads/stores might cause dependency between integer and FP issue:
  - Replace load reservation station with a load queue; operands must be read in the order they are fetched
  - Load checks addresses in Store Queue to avoid RAW violation
  - Store checks addresses in Load Queue to avoid WAR, WAW

Hardware-Based Speculation

- Trying to exploit more ILP while maintaining control dependencies becomes a burden
- Overcome control dependencies by **speculating** on the outcome of branches and executing the program as if our guesses were correct
  - Need to handle incorrect guesses
- Key ideas:
  - Dynamic branch prediction
  - Speculation
  - Dynamic scheduling

Implementing Speculation

- We consider building on top of Tomasulo's algorithm
  - Must separate bypassing of results among instructions from actual completion (**write-back**) of instructions
  - Cannot allow updates to be performed that can't be undone
  - Instruction commit updates register or memory when instruction no longer speculative
  - Need to add re-order buffer
- Key idea: execute out-of-order but commit in-order
Tomasulo extended to handle speculation

Reorder buffer

- Contains 4 fields:
  1. Instruction type indicates whether branch, store, or register op
  2. Destination field memory or register
  3. Value field
  4. Ready flag indicates instruction has completed operation

- The renaming function of the reservation stations is replaced by the ROB
- Every instruction has a ROB entry until it commits
  - Therefore tag results using ROB entry number

Instruction execution

1. Issue - get instruction from instruction Q and issue if reservation station and ROB slots available - sometimes called dispatch
2. Execute - when both operands available at the reservation station - sometimes called issue
3. Write result - when result available, write to CDB tagged by ROB entry #; mark reservation station slot available
4. Commit - when instruction at head of Q ready, writeback result unless mispredicted branch. In latter case, flush all remaining instructions in ROB and commence fetching at target.

Register renaming, virtual registers versus Reorder Buffers

- Alternative to Reorder Buffer is a larger virtual set of registers and register renaming
- Virtual registers hold both architecturally visible registers + temporary values
  - replace functions of reorder buffer and reservation station
- Renaming process maps names of architectural registers to registers in virtual register set
  - Changing subset of virtual registers contains architecturally visible registers
- Simplifies instruction commit: mark register as no longer speculative, free register with old value
- Adds 40-80 extra registers: Alpha, Pentium,...
  - Size limits no. instructions in execution (used until commit)
How much to speculate?

- Speculation Pro: uncover events that would otherwise stall the pipeline (cache misses)
- Speculation Con: speculate costly if exceptional event occurs when speculation was incorrect
- Typical solution: speculation allows only low-cost exceptional events (1st-level cache miss)
- When expensive exceptional event occurs, (2nd-level cache miss or TLB miss) processor waits until the instruction causing event is no longer speculative before handling the event
- Assuming single branch per cycle: future may speculate across multiple branches!

Limits to ILP

Initial HW Model here; MIPS compilers.
Assumptions for ideal/perfect machine to start:
1. **Register renaming** - infinite virtual registers
   => all register WAW & WAR hazards are avoided
2. **Branch prediction** - perfect; no mispredictions
3. **Jump prediction** - all jumps perfectly predicted
2 & 3 => machine with perfect speculation & an unbounded buffer of instructions available
4. **Memory-address alias analysis** - addresses are known & a store can be moved before a load provided addresses not equal
   
   Also:
   - unlimited number of instructions issued/clock cycle;
   - perfect caches;
   - 1 cycle latency for all instructions (FP *, /);

Upper Limit to ILP: Ideal Machine

(Figure 3.34, page 294)

<table>
<thead>
<tr>
<th>Programs</th>
<th>Integer</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>54.8</td>
<td></td>
</tr>
<tr>
<td>espresso</td>
<td>62.6</td>
<td></td>
</tr>
<tr>
<td>li</td>
<td>17.9</td>
<td></td>
</tr>
<tr>
<td>fpppp</td>
<td>75.2</td>
<td></td>
</tr>
<tr>
<td>dodued</td>
<td>118.7</td>
<td></td>
</tr>
<tr>
<td>tomcatv</td>
<td>150.1</td>
<td></td>
</tr>
</tbody>
</table>

IPC

Integer: 18 - 60
FP: 75 - 150
More Realistic HW: Branch Impact

Change from Infinite window to examine to 2000 and maximum issue of 64 instructions per clock cycle

FP: 15 - 45

Integer: 6 - 12

Program

Perfect  Tournament  BHT (512)  Profile  No prediction

More Realistic HW: Renaming Register Impact

Change 2000 instr window, 64 instr issue, 8K 2 level Prediction

FP: 11 - 45

Integer: 5 - 15

Program

Infinite  256  128  64  32  None

More Realistic HW: Memory Address Alias Impact

Change 2000 instr window, 64 instr issue, 8K 2 level Prediction, 256 renaming registers

FP: 4 - 45 (Fortran, no heap)

Integer: 4 - 9

Program

Perfect  Global/Stack perf; Inspec.  None  heap conflicts

Realistic HW for '00: Window Impact

(Figure 3.45, Page 309)

Perfect disambiguation (HW), 1K Selective Prediction, 16 entry return, 64 registers, issue as many as window

FP: 8 - 45

Integer: 6 - 12

Program

Infinite  256  128  64  32  16  8  4
How to Exceed ILP Limits of this study?

- WAR and WAW hazards through memory: eliminated WAW and WAR hazards through register renaming, but not in memory usage
- Unnecessary dependences (compiler not unrolling loops so iteration variable dependence)
- Overcoming the data flow limit: value prediction, predicting values and speculating on prediction
  - Address value prediction and speculation predicts addresses and speculates by reordering loads and stores; could provide better aliasing analysis, only need predict if addresses =


- Max issue: 4 instructions (many CPUs)
- Max rename registers: 128 (Pentium 4)
- Max BHT: 4K x 9 (Alpha 21264B), 16Kx2 (Ultra III)
- Max Window Size (OOO): 126 instructions (Pentium 4)
- Max Pipeline: 22/24 stages (Pentium 4)

Conclusion

- 1985-2000: 1000X performance
  - Moore’s Law transistors/chip => Moore’s Law for Performance/MPU
- Hennessy: industry been following a roadmap of ideas known in 1985 to exploit Instruction Level Parallelism and (real) Moore’s Law to get 1.55X/year
  - Caches, Pipelining, Superscalar, Branch Prediction, Out-of-order execution, ...
- ILP limits: To make performance progress in future need to have explicit parallelism from programmer vs. implicit parallelism of ILP exploited by compiler, HW?
  - Otherwise drop to old rate of 1.3X per year?
  - Less than 1.3X because of processor-memory performance gap?
- Impact on you: if you care about performance, better think about explicitly parallel algorithms vs. rely on ILP?