Multiple-Issue Processors: Motivation

- Ideal processor: CPI of 1
  - no hazards, 1-cycle memory latency

- Realistic processor: CPI ~1
  - Dynamic scheduling – avoids WAR & WAW dependencies
  - Branch prediction – avoids control flow dependencies
  - Caches – minimize AMAT

- Question: can we do better than that???
Multiple-Issue Processors

- Answer: Yes!
  - start more than one instruction in the same clock cycle
  - CPI < 1 (or IPC > 1, Instructions per Cycle)

- Two approaches:
  - Superscalar: instructions are chosen dynamically by the hardware
  - VLIW (Very Long Instruction Word): instructions are chosen statically by the compiler (and assembled in a single long “instruction”)
Superscalar Processors

- Hardware attempts to issue up to $n$ instructions on every cycle, where $n$ is the issue width of the processor and the processor is said to have $n$ issue slots and to be a $n$-wide processor.
- Instructions issued must respect data dependences.
- In some cycles not all issue slots can be used.
- Extra hardware is needed to detect more combinations of dependences and hazards and to provide more bypasses.
- Branches?
  - With branch prediction, we can predict branches and fetch instructions.
  - Can we execute such predicted instructions?
Speculative Execution

- **Speculative execution** – *execute* control dependent instructions even when we are not sure if they should be executed

- Hardware undo, in case of a misprediction
  - Software recovery too costly, performance-wise

- Tomasulo + multi-issue + speculation
  - Implemented in many current processors

- Key Idea: Execute out-of-order but **commit** in order
  - Commit: the results and side-effects (e.g., flags, exceptions) of an instruction are made visible to the rest of the system
Extending Tomasulo to Support Speculation

- Approach: buffer result until instruction ready to commit (i.e., known to be non-speculative)
  - Use buffered result for forwarding to dependent instructions
  - Discard buffered result if the instruction is on a mis-speculated execution path
  - At commit, write buffered result to register or memory

- Decouples forwarding (potentially speculative) from update of architecturally-visible state (non-speculative)
  - Architecturally visible state: registers (R0-Rn, F0-Fn, memory)
Enabling Speculation with the Reorder Buffer

New structure: Reorder Buffer (ROB)

- Holds completed results until commit time
- Organized as a queue ordered by program (i.e., fetch) order
- Takes over the role of the reservation stations for tracking dependencies and bypassing values
  - Accessed by dependent instructions for forwarding of completed, but not-yet-committed, results
  - Reservation stations still needed to hold issued instructions until they begin execution
- Flushed once mis-speculation is discovered (mispredicted branch commits)
- Enables precise exceptions
  - Exception state recorded in ROB
  - Flushed if exception occurred on a mis-predicted path
Tomasulo with Hardware Speculation

- **Issue:**
  - Get instruction from queue
  - Issue if an RS is free and an ROB entry is also free
  - Stall if no RS or no free ROB entry
  - Instructions now tagged with ROB entry number, not RS.Id

- **Execute:**
  - Same as before: monitor CDB and start instruction when operands are available

- **Write Result:**
  - CDB broadcasts result with ROB identifier
  - ROB captures result to commit later
  - Store operations also saved in the ROB until store data is available and store instruction is committed

- **Commit:**
  - If branch, check prediction and squash following instructions if incorrect
  - If store, send data and address to memory unit and perform write action
  - Else, update register with new value and release ROB entry
VLIW Processors

- Compiler chooses and “packs” independent instructions into a single long “instruction” word or “bundle”
- No need for hardware to check the instructions for dependences
- Not all portions of the long instruction word will be used in every cycle
- Compiler must be able to expose a lot of parallelism in the instruction flow
- Example:

<table>
<thead>
<tr>
<th>MEM op 1</th>
<th>MEM op 2</th>
<th>FP op 1</th>
<th>FP op 2</th>
<th>INT op</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld f18,-32(r1)</td>
<td>ld f22,-40(r1)</td>
<td>addd f4,f0,f2</td>
<td>addd f8,f6,f2</td>
<td></td>
</tr>
</tbody>
</table>
Superscalar vs. VLIW Processors

**SuperScalar**

+ Able to handle dynamic events like cache misses, unpredictable memory dependences, branches, etc.
+ Can exploit old binaries from previous implementations
- Complexity limits issue width to 4-8

**VLIW**

+ Much simpler hardware implementation
+ Implementations can have wider issue than superscalars
- Require more complex compiler support
- Cannot use old binaries when pipeline implementation changes
- Code size increases because of empty issue slots
What are the practical Limitations to ILP?

- Limitations on max issue width and instruction window size
- Effects of realistic branch prediction
- The effect of limited numbers of rename registers
- Memory aliasing
- Variable memory latencies (because of caches)

H&P 4th ed, fig 3.1 (p.157) shows the available ILP in a perfect processor, with none of the above constraints. This is shown for 6 of the SPEC92 benchmarks – the first 3 are integer, the final 3 are floating point benchmarks. These levels of ILP are impossible to achieve in practice, due to the limitations on window size, issue width, branch prediction accuracy and cache performance.
Effect of Instruction Window (i.e., ROB)

Instructions Per Clock

H&P 4th ed, fig 3.2 (p.159)
Effect of Branch prediction

![Graph showing the effect of branch prediction on instruction issues per cycle for different programs and prediction methods.](image)

- Program: gcc, espresso, li, fpppp, doducd, tomcatv
- Prediction methods: Perfect, Selective predictor, Standard 2-bit, Static, None
- Instructions issues per cycle: 0, 10, 20, 30, 40, 50, 60

*H&P 4th ed, fig 3.3 (p.160)*
Effect of Limited Rename Registers

![Bar chart showing instruction issues per cycle for different programs and register sizes.](chart.png)

**Programs:**
- gcc
- espresso
- li
- fpppp
- doducd
- tomcatv

**Register Sizes:**
- Infinite
- 256
- 128
- 64
- 32
- None

**Instruction Issues per Cycle:**
- gcc
  - Infinite: 11
  - 256: 10
  - 128: 8.7
  - 64: 4.9
  - 32: 4.3
  - None: 4.3

- espresso
  - Infinite: 15
  - 256: 15
  - 128: 9.8
  - 64: 4.7
  - 32: 4
  - None: 4

- li
  - Infinite: 12
  - 256: 12
  - 128: 10.9
  - 64: 5.8
  - 32: 5.2
  - None: 5.2

- fpppp
  - Infinite: 59
  - 256: 35
  - 128: 20.4
  - 64: 5.4
  - 32: 3.5
  - None: 3.5

- doducd
  - Infinite: 16
  - 256: 15
  - 128: 11
  - 64: 5.3
  - 32: 4.7
  - None: 4.7

- tomcatv
  - Infinite: 45
  - 256: 44
  - 128: 27.7
  - 64: 6.6
  - 32: 4.9
  - None: 4.9

*H&P 4th ed, fig 3.5 (p.163)*
Limits to Multiple-issue

- Fundamental limits to ILP in most programs:
  - Need $N$ independent instructions to keep a $W$-issue processor busy, where $N = W \times \text{pipeline depth}$
  - Data and control dependences significantly limit amount of ILP

- Complexity of the hardware based on issue width:
  - Number of functional units increases linearly → OK
  - Number of ports for register file increases linearly → bad
  - Number of ports for memory increases linearly → bad
  - Number of dependence tests increases quadratically → bad
  - Bypass/forwarding logic and wires increases quadratically → bad

These two tend to ultimately limit the width of practical dynamically-scheduled superscalars
Summary of Factors Limiting ILP in Real Programs

- Compared with an ideal processor
  - Limited instruction window
  - Finite number of registers (introduces WAW and WAR stalls)
  - Imperfect branch prediction (pipeline flushes)
  - Limited issue width
  - Instruction fetch delays (cache misses)
  - Imperfect memory disambiguation (conservative RAW stalls)

- Implications for future performance growth?
  - Single processor has inherent limits
  - To use future silicon area, need to go to multiple processors