## Evolution of Processor Performance

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistor Count</td>
<td>2K-100K</td>
<td>100K-1M</td>
<td>1M-100M</td>
<td>100M-1B</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>0.1-3MHz</td>
<td>3-30MHz</td>
<td>30M-1GHz</td>
<td>1-15GHz</td>
</tr>
<tr>
<td>Instruction/Cycle</td>
<td>&lt; 0.1</td>
<td>0.1-0.9</td>
<td>0.9-1.9</td>
<td>1.9-2.9 (?)</td>
</tr>
<tr>
<td>CPI</td>
<td>&gt;10</td>
<td>1.1-10</td>
<td>0.5-1.1</td>
<td>.35-.5 (?)</td>
</tr>
</tbody>
</table>

Source: John P. Chen, Intel Labs
Multiple Instruction Issue: CPI < 1

- To improve a pipeline’s CPI to be better [less] than one, and to utilize ILP better, a number of independent instructions have to be issued in the same pipeline cycle.

- Multiple instruction issue processors are of two types:
  - **Superscalar:** A number of instructions (2-8) is issued in the same cycle, scheduled statically by the compiler or dynamically (Tomasulo).
    - PowerPC, Sun UltraSparc, Alpha, HP 8000 ...
  - **VLIW** (Very Long Instruction Word):
    A fixed number of instructions (3-6) are formatted as one long instruction word or packet (statically scheduled by the compiler).
    - Joint HP/Intel agreement (Itanium, Q4 2000).
    - Intel Architecture-64 (IA-64) 64-bit address:
      - Explicitly Parallel Instruction Computer (EPIC): Itanium.

- Limitations of the approaches:
  - Available ILP in the program (both).
  - Specific hardware implementation difficulties (superscalar).
  - VLIW optimal compiler design issues.
Multiple Instruction Issue:

Superscalar Vs. VLIW

- Smaller code size.
- Binary compatibility across generations of hardware.
- Simplified Hardware for decoding, issuing instructions.
- No Interlock Hardware (compiler checks?)
- More registers, but simplified hardware for register ports.
**Simple Statically Scheduled Superscalar Pipeline**

- Two instructions can be issued per cycle (two-issue superscalar).
- One of the instructions is integer (including load/store, branch). The other instruction is a floating-point operation.
  - This restriction reduces the complexity of hazard checking.
- Hardware must fetch and decode two instructions per cycle.
- Then it determines whether zero (a stall), one or two instructions can be issued per cycle.

<table>
<thead>
<tr>
<th>Instruction Type</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>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

Two-issue statically scheduled pipeline in operation
FP instructions assumed to be adds
Intel/HP VLIW “Explicitly Parallel Instruction Computing (EPIC)”

- Three instructions in 128 bit “Groups”; instruction template fields determines if instructions are dependent or independent
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show dependencies of more than three instructions.
- 128 integer registers + 128 floating point registers
  - No separate register files per functional unit as in old VLIW.
- Hardware checks dependencies
  (interlocks ⇒ binary compatibility over time)
- Predicated execution: An implementation of conditional instructions used to reduce the number of conditional branches used in the generated code ⇒ larger basic block size
- IA-64: Name given to instruction set architecture (ISA).
- Itanium: Name of the first implementation (2001).
Intel/HP EPIC VLIW Approach

original source code

Exposé Instruction Parallelism

compiler

Instruction Dependency Analysis

Optimize

Exploit Parallelism: Generate VLIWs

Instruction 2
41 bits

Instruction 1
41 bits

Instruction 0
41 bits

Template
5 bits

128-bit bundle

EECC551 - Shaaban
# IA-64 Instruction Types

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Description</th>
<th>Execution Unit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Integer ALU</td>
<td>I-unit or M-unit</td>
</tr>
<tr>
<td>I</td>
<td>Non-integer ALU</td>
<td>I-unit</td>
</tr>
<tr>
<td>M</td>
<td>Memory</td>
<td>M-unit</td>
</tr>
<tr>
<td>F</td>
<td>Floating Point</td>
<td>F-unit</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>B-unit</td>
</tr>
<tr>
<td>L+X</td>
<td>Extended</td>
<td>I-unit/B-unit</td>
</tr>
</tbody>
</table>
IA-64 Template Use

• The template specifies the functional units for the three operations in the instruction.

• Possible instruction combinations:
  – M-unit, I-unit, I-unit
  – M-unit, L-unit, X-unit
  – M-unit, M-unit, I-unit
  – M-unit, F-unit, I-unit
  – M-unit, M-unit, F-unit
  – M-unit, I-unit, B-unit
  – M-unit, B-unit, B-unit
  – B-unit, B-unit, B-unit
  – M-unit, M-unit, B-unit
  – M-unit, F-unit, B-unit
  – M-unit, F-unit, B-unit
IA-64 Instruction Format Example:
A-Unit Instruction Format

Register-register format:

<table>
<thead>
<tr>
<th>40</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>20</th>
<th>19</th>
<th>13</th>
<th>12</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X2a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 8 is the major opcode for this instruction type.
- X2a, X2b, Ve, and X4 are opcode extensions.
- qp is the predicate register assigned to this operation.
IA-64 Instruction Format Example:
A-Unit Instruction Format

14-bit Immediate format:

<table>
<thead>
<tr>
<th>40</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>27</th>
<th>26</th>
<th>20</th>
<th>19</th>
<th>13</th>
<th>12</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>s</td>
<td>X2a</td>
<td>Ve</td>
<td>imm6d</td>
<td>R3</td>
<td>imm7b</td>
<td>R1</td>
<td>qp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- X2a and Ve are opcode extensions.
- qp is the predicate register assigned to this operation.
- The immediate value is made up of s, imm6d, and imm7b.
- Other formats available for 8-bit and 22-bit immediates.
IA-64 Instruction Format Example:
M-Unit Instruction Format

Integer Load/Store operations:

<table>
<thead>
<tr>
<th>40</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>20</th>
<th>19</th>
<th>13</th>
<th>12</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>m</td>
<td>X6</td>
<td>hint</td>
<td>x</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td>qp</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- m and x determine the subset of load/store operations.
- Bits 35:32 give special directives for LAT, cache, etc.
- Bits 31:30 tell the size of the load (1, 2, 4, or 8 bytes).
- Hint allows for system pre-fetching of data by specifying if temporal locality exists for this data.
IA-64 Instruction Format Example:
B-Unit Instruction Format

IP-relative branch:

<table>
<thead>
<tr>
<th>40</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>26</th>
<th>20</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>s</td>
<td>d</td>
<td>wh</td>
<td>imm(_{20b})</td>
<td>p</td>
<td>btype</td>
<td>qp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>20</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Opcode 0- indirect branch, 1- indirect call, 4- IP-relative branch, 5- IP-relative call
- btype sub-category of branch type, qp- branch condition
- d- cache de-allocation, wh- branch hint, s & imm\(_{20b}\) give offset
- p pre-fetch 0-few, or 1-many inst. following the target
Unrolled Loop Example for Scalar Pipeline

1. Loop: L.D F0,0(R1)
2. L.D F6,-8(R1)
3. L.D F10,-16(R1)
4. L.D F14,-24(R1)
5. ADD.D F4,F0,F2
6. ADD.D F8,F6,F2
7. ADD.D F12,F10,F2
8. ADD.D F16,F14,F2
9. S.D F4,0(R1)
10. S.D F8,-8(R1)
11. DADDUI R1,R1,#-32
12. S.D F12,-16(R1)
13. BNE R1,R2,LOOP
14. S.D F16,8(R1) ; 8-32 = -24

14 clock cycles, or 3.5 per iteration
Loop Unrolling in Superscalar Pipeline:
(1 Integer, 1 FP/Cycle)

<table>
<thead>
<tr>
<th>Integer instruction</th>
<th>FP instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F6,-8(R1)</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>3</td>
</tr>
<tr>
<td>L.D F14,-24(R1)</td>
<td>ADD.D F8,F6,F2</td>
<td>4</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>5</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>ADD.D F16,F14,F2</td>
<td>6</td>
</tr>
<tr>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F20,F18,F2</td>
<td>7</td>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>DADDUI R1,R1,#-40</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>S.D F16,-24(R1)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>BNE R1,R2,LOOP</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration (1.5X)
- 7 issue slots wasted
### Loop Unrolling in VLIW Pipeline

*(2 Memory, 2 FP, 1 Integer / Cycle)*

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td>4</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>S.D F8, -8(R1)</td>
<td>ADD.D F28,F26,F2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>S.D F12, -16(R1)</td>
<td>S.D F16,-24(R1)</td>
<td></td>
<td></td>
<td>DADDUI R1,R1,#-56</td>
<td>6</td>
</tr>
<tr>
<td>S.D F20, 24(R1)</td>
<td>S.D F24,16(R1)</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>S.D F28, 8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>BNE R1,R2,LOOP</td>
<td>8</td>
</tr>
</tbody>
</table>

*Unrolled 7 times to avoid delays*

*7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)*

*Average: 2.5 ops per clock, 50% efficiency*

*Note: Needs more registers in VLIW (15 vs. 6 in Superscalar)*
Superscalar Dynamic Scheduling

• How to issue two instructions and keep in-order instruction issue for Tomasulo?
  – Assume: 1 integer + 1 floating-point operations.
  – 1 Tomasulo control for integer, 1 for floating point.
• Issue at 2X Clock Rate, so that issue remains in order.
• Only FP loads might cause a 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.
• Called “Decoupled Architecture”
Multiple Instruction Issue Challenges

• While a two-issue single Integer/FP split is simple in hardware, we get a CPI of 0.5 only for programs with:
  – Exactly 50% FP operations
  – No hazards of any type.

• If more instructions issue at the same time, greater difficulty of decode and issue operations arise:
  – Even for a 2-issue superscalar machine, we have to examine 2 opcodes, 6 register specifiers, and decide if 1 or 2 instructions can issue.

• VLIW: tradeoff instruction space for simple decoding
  – The long instruction word has room for many operations.
  – By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  – E.g. 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    • 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  – Need compiling technique that schedules across several branches.
Limits to Multiple Instruction Issue Machines

• Inherent limitations of ILP:
  – If 1 branch exist for every 5 instruction: How to keep a 5-way VLIW busy?
  – Latencies of unit adds complexity to the many operations that must be scheduled every cycle.
  – For maximum performance multiple instruction issue requires about:

    Pipeline Depth × No. Functional Units
    independent instructions per cycle.

• Hardware implementation complexities:
  – Duplicate FUs for parallel execution are needed.
  – More instruction bandwidth is essential.
  – Increased number of ports to Register File (datapath bandwidth):
    • VLIW example needs 7 read and 3 write for Int. Reg.
    & 5 read and 3 write for FP reg
  – Increased ports to memory (to improve memory bandwidth).
  – Superscalar decoding complexity may impact pipeline clock rate.
Superscalar Architectures:
Issue Slot Waste Classification

- Empty or wasted issue slots can be defined as either vertical waste or horizontal waste:
  - Vertical waste is introduced when the processor issues no instructions in a cycle.
  - Horizontal waste occurs when not all issue slots can be filled in a cycle.

![Diagram showing issue slots and waste classification]

- Full issue slot
- Empty issue slot

- Horizontal waste = 9 slots
- Vertical waste = 12 slots
Sources of Unused Issue Cycles in an 8-issue Superscalar Processor.

Processor busy represents the utilized issue slots; all others represent wasted issue slots.

61% of the wasted cycles are vertical waste, the remainder are horizontal waste.

Workload: SPEC92 benchmark suite.

Superscalar Architectures:

All possible causes of wasted issue slots, and latency-hiding or latency reducing techniques that can reduce the number of cycles wasted by each cause.

<table>
<thead>
<tr>
<th>Source of Wasted Issue Slots</th>
<th>Possible Latency-Hiding or Latency-Reducing Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction tlb miss, data tlb miss</td>
<td>decrease the TLB miss rates (e.g., increase the TLB sizes); hardware instruction prefetching; hardware or software data prefetching; faster servicing of TLB misses</td>
</tr>
<tr>
<td>I cache miss</td>
<td>larger, more associative, or faster instruction cache hierarchy; hardware instruction prefetching</td>
</tr>
<tr>
<td>D cache miss</td>
<td>larger, more associative, or faster data cache hierarchy; hardware or software prefetching; improved instruction scheduling; more sophisticated dynamic execution</td>
</tr>
<tr>
<td>branch misprediction</td>
<td>improved branch prediction scheme; lower branch misprediction penalty</td>
</tr>
<tr>
<td>control hazard</td>
<td>speculative execution; more aggressive if-conversion</td>
</tr>
<tr>
<td>load delays (first-level cache hits)</td>
<td>shorter load latency; improved instruction scheduling; dynamic scheduling</td>
</tr>
<tr>
<td>short integer delay</td>
<td>improved instruction scheduling</td>
</tr>
<tr>
<td>long integer, short fp, long fp delays</td>
<td>(multiply is the only long integer operation, divide is the only long floating point operation) shorter latencies; improved instruction scheduling</td>
</tr>
<tr>
<td>memory conflict</td>
<td>(accesses to the same memory location in a single cycle) improved instruction scheduling</td>
</tr>
</tbody>
</table>

Hardware Support for Extracting More Parallelism

- Compiler ILP techniques (loop-unrolling, software Pipelining etc.) are not effective to uncover maximum ILP when branch behavior is not well known at compile time.

- Hardware ILP techniques:
  - **Conditional or Predicted Instructions**: An extension to the instruction set with instructions that turn into no-ops if a condition is not valid at run time.
  - **Speculation**: An instruction is executed before the processor knows that the instruction should execute to avoid control dependence stalls:
    - **Static Speculation** by the compiler with hardware support:
      - The compiler labels an instruction as speculative and the hardware helps by ignoring the outcome of incorrectly speculated instructions.
      - Conditional instructions provide limited speculation.
    - **Dynamic Hardware-based Speculation**:
      - Uses dynamic branch-prediction to guide the speculation process.
      - Dynamic scheduling and execution continued passed a conditional branch in the predicted branch direction.
Conditional or Predicted Instructions

- Avoid branch prediction by turning branches into conditionally-executed instructions:

  \[
  \text{if} \ (x) \ \text{then} \ (A = B \ \text{op} \ C) \ \text{else} \ \text{NOP}
  \]

  - If false, then neither store result nor cause exception: instruction is annulled (turned into NOP).
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move.
  - HP PA-RISC can annul any following instruction.
  - IA-64: 64 1-bit condition fields selected so conditional execution of any instruction (Predication).

- Drawbacks of conditional instructions
  - Still takes a clock cycle even if “annulled”.
  - Must stall if condition is evaluated late.
  - Complex conditions reduce effectiveness; condition becomes known late in pipeline.
IA-64 Predication Example

Pseudo Code

If (a==b)
    jmp 2
else
    jmp 3
2:
callfunction1;
callfunction2;
jmp 4:
3:
callfunction3;
callfunction4;
4:

Branch in traditional Architecture

PREDICATION

Predicate True  Predicate False
For below   For below
jmp 2       jmp 3
     callfunction1  callfunction3
callfunction2  callfunction4

Branches
Eliminated
With Predication

Predication will evaluate True and False at the same time and then select the branch to be used after the comparison is evaluated.
Dynamic Hardware-Based Speculation

- Combines:
  - Dynamic hardware-based branch prediction
  - Dynamic Scheduling: of multiple instructions to issue and execute out of order.

- Continue to dynamically issue, and execute instructions passed a conditional branch in the dynamically predicted branch direction, before control dependencies are resolved.
  - This overcomes the ILP limitations of the basic block size.
  - Creates dynamically speculated instructions at run-time with no compiler support at all.
  - If a branch turns out as mispredicted all such dynamically speculated instructions must be prevented from changing the state of the machine (registers, memory).

- Addition of commit (retire or re-ordering) stage and forcing instructions to commit in their order in the code (i.e., to write results to registers or memory).
- Precise exceptions are possible since instructions must commit in order.
Hardware-Based Speculation

Speculative Execution + Tomasulo’s Algorithm
Four Steps of Speculative Tomasulo Algorithm

1. Issue — Get an instruction from FP Op Queue
   If a reservation station and a reorder buffer slot are free, issue instruction & send operands & reorder buffer number for destination (this stage is sometimes called “dispatch”)

2. Execution — Operate on operands (EX)
   When both operands are ready then execute; if not ready, watch CDB for result; when both operands are in reservation station, execute; checks RAW (sometimes called “issue”)

3. Write result — Finish execution (WB)
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. Commit — Update registers, memory with reorder buffer result
   – When an instruction is at head of reorder buffer & the result is present, update register with result (or store to memory) and remove instruction from reorder buffer.
   – A mispredicted branch at the head of the reorder buffer flushes the reorder buffer (sometimes called “graduation”)

⇒ Instructions issue in order, execute (EX), write result (WB) out of order, but must commit in order.
Advantages of HW (Tomasulo) vs. SW (VLIW) Speculation

- HW determines address conflicts.
- HW provides better branch prediction.
- HW maintains precise exception model.
- HW does not execute bookkeeping instructions.
- Works across multiple implementations
- SW speculation is much easier for HW design.