Multiple Instruction Issue: CPI < 1

- To improve a pipeline’s CPI to be 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) are 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).

- Both types are limited by:
  - Available ILP in the program.
  - Specific hardware implementation difficulties.

### Multiple Issue Processors

<table>
<thead>
<tr>
<th>Name</th>
<th>Issue</th>
<th>Hazard</th>
<th>Sched.</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar (Static)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Static</td>
<td>Sun UltraSPARC II/III</td>
</tr>
<tr>
<td>Superscalar (Dynamic)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic</td>
<td>IBM Power2</td>
</tr>
<tr>
<td>Superscalar (Speculative)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic With speculation</td>
<td>Pentium II/4, MIPS R10K, Alpha 21264, HP PA 8500 IBM RS64III</td>
</tr>
<tr>
<td>VLIW</td>
<td>Static</td>
<td>Software</td>
<td>Static</td>
<td>Trimedia, i860</td>
</tr>
<tr>
<td>EPIC</td>
<td>Mostly Static</td>
<td>Mostly Static</td>
<td>Mostly Static</td>
<td>Itanium</td>
</tr>
</tbody>
</table>
Multiple Instruction Issue:

Superscalar Vs. VLIW

- Complex hardware (Tomasulo’s, predictors, etc.)
- Smaller code size.
- Binary compatibility across generations of hardware.
- Simpler compiler
- Fewer registers
- Simplified Hardware for decoding, issuing instructions.
- Greater code size
- Lack of binary compatibility within generation
- Complex compiler
- Greater no. of registers

Superscalar (Static) Pipeline Operation

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Pipe stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer instruction</td>
<td>IF</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF</td>
</tr>
<tr>
<td>Integer instruction</td>
<td>IF</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF</td>
</tr>
<tr>
<td>Integer instruction</td>
<td>IF</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF</td>
</tr>
<tr>
<td>Integer instruction</td>
<td>IF</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF</td>
</tr>
</tbody>
</table>

Fig. 3.24: Two instructions issued at the same time - one integer, one FP
Superscalar (Static)

- Multiple instruction fetch is complex - what if instructions come from more than one cache block!
- Issue: the fetched instructions are the “issue packet”
- Hazard detection done in parallel (standard hazards) $O(n^2 - n)$ to detect hazard for $n$-issue.
- Issue those that can proceed in order.
- Few stalls because integer and FP instructions use separate register sets.
- FP unit is pipelined to realize gains from multiple issue.

Superscalar (Static)

- Issue stage is so complex it is itself pipelined into two stages
- Stage 1: how many instructions from within this packet can issue based on internal hazards
- Stage 2: examine hazards among the selected instructions and previously issued instructions
Intel/HP VLIW “Explicitly Parallel Instruction Computing (EPIC)”

- Three instructions in 128 bit “Groups”; instruction template fields determines if instructions are dependent or independent
- 128 integer registers + 128 floating point registers
  - No separate register files per functional unit as in old VLIW.
- Hardware checks dependencies
- 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 is the instruction architecture
- Itanium is the first machine, released in 2000?

Intel/HP EPIC VLIW Approach

original source code

Expose Instruction Parallelism

compiler

Optimize

Instruction Dependency Analysis

Exploit Parallelism: Generate VLIWs

128-bit bundle

Instruction 2  Instruction 1  Instruction 0  Template
Unrolled Loop Example for Scalar Pipeline

1 Loop:

1. LD F0,0(R1)
2. LD F6,-8(R1)
3. LD F10,-16(R1)
4. LD F14,-24(R1)
5. ADDD F4,F0,F2
6. ADDD F8,F6,F2
7. ADDD F12,F10,F2
8. ADDD F16,F14,F2
9. SD 0(R1),F4
10. SD -8(R1),F8
11. SD -16(R1),F12
12. SUBI R1,R1,#32
13. BNEZ R1,LOOP
14. SD 8(R1),F16  ; 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>LD F0,0(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F6,-8(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>2</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>ADDD F8,F6,F2</td>
<td>3</td>
</tr>
<tr>
<td>LD F14,-24(R1)</td>
<td>ADDD F12,F10,F2</td>
<td>4</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>ADDD F16,F14,F2</td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>ADDD F20,F18,F2</td>
<td>6</td>
</tr>
<tr>
<td>SD -8(R1),F8</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>SD -24(R1),F16</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>SUBI R1,R1,#40</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>BNEZ R1,LOOP</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

• 3 cycles for FP
• Unrolled 5 times to avoid delays (+1 due to SS)
• 12 clocks, or 2.4 clocks per iteration (1.5X)
Loop Unrolling in VLIW Pipeline (2 Memory, 2 FP, 1 Integer / Cycle)

<table>
<thead>
<tr>
<th></th>
<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>LD F0,0(R1)</td>
<td>LD F6,-8(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>ADDD F8,F6,F2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>LD F14,-24(R1)</td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F16,F14,F2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADDD F20,F18,F2</td>
<td>ADDD F24,F22,F2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>SD -8(R1),F8</td>
<td>ADDD F28,F26,F2</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>SD -24(R1),F16</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td>SD -40(R1),F24</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD -40(R1),F24</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays (why not 8?)
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.
  – Register file: need 2X reads and 1X writes per cycle
  – Rename logic: must be able to rename same register multiple times in one cycle.
  – Result buses must complete multiple instructions per cycle ==> multiple buses and lots of associative (matching) logic for reservation stations.

Multiple Instruction Issue Challenges

• 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 exists for every 5 instructions: 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:
  
  \[
  \text{Pipeline Depth} \times \text{No. Functional Units} \times \text{independent instructions per cycle.}
  \]

**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.

Hardware Support for Extracting More Parallelism:

**HARDWARE SPECULATION**

- 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 past a conditional branch in the predicted branch direction.
Dynamic Hardware-Based Speculation

- Combines:
  - Dynamic hardware-based branch prediction
  - Dynamic Scheduling: of multiple instructions to issue and execute out of order, e.g. using Tomasulo’s Algorithm

- Continue to dynamically issue, and execute instructions past 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.

Dynamic Hardware-Based Speculation

- If a branch turns out as mispredicted all such dynamically speculated instructions must be prevented from changing the state of the machine (registers, memory).
- Key idea: Addition of commit (retire or re-ordering) stage separate from execute stage.
- Forcing instructions to commit in their order in the code (i.e. to write results to registers or memory).
Hardware Speculation Implementation: Re-order Buffer (ROB)

Buffer that holds instructions that have executed but not committed.

When it is discovered that the speculations was wrong, these instructions are deleted!

Exceptions are similarly buffered and not handled until instruction is committed.

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>destination</th>
<th>Value of result</th>
<th>Ready bit</th>
</tr>
</thead>
</table>

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, execute (EX), write result (WB) out of order but must commit in order.