VLIW Processors

- Very Long Instruction Word
- N-wide VLIW issues packets of N instructions simultaneously. Compiler guarantees independence of those N instructions.

```
add r5, r4, r1 muld f6, f4, f2 lw r2, 0(r7) sub r8, r6, r1 beqz r9, label
sub r11, r5, r1 addd f8, f0, f6 sw r5, 8(r7) nop beqz r2, l2
```

Loop Unrolling in VLIW

```
Memory reference 1 Memory reference 2 FP operation 1 FP op. 2 Int. op/branch Clock
LD F0(0(R1))  LD F6,-8(R1) ADDD F4,F0,F2 ADDD F8,F6,F2 1
LD F10,-16(R1) LD F14,-24(R1) ADDD F12,F10,F2 ADDD F16,F14,F2 2
LD F18,-32(R1) LD F22,-40(R1) ADDD F20,F18,F2 ADDD F24,F22,F2 3
LD F26,-48(R1) ADDD F28,F26,F2 4
LD 0(R1),F4 SD 0(R1),F4 SUBI R1,R1,#48 5
SD -16(R1),F12 SD -24(R1),F16 BNEZ R1,LOOP 6
SD -32(R1),F20 SD -40(R1),F24 7
SD -0(R1),F28
```

- Unrolled 7 times to avoid delays
- 7 results in 9 clocks, or 1.3 clocks per iteration
- Need more registers in VLIW

Limits to Multi-Issue Machines

- 1 branch in 5 instructions => how to keep a 5-way VLIW busy?
- Latencies of units => many operations must be scheduled
  - Need about Pipeline Depth x No. Functional Units of independent operations to keep machines busy
- Instruction mix may not match hardware mix
  - Need duplicate FUs, increased flexibility
- Increase ports to Register File (VLIW example needs 6 read and 3 write for Int. Reg. & 6 read and 4 write for FP reg)
- Increase ports to memory
- Decoding SS and impact on clock rate

Superscalar vs. VLIW

- Superscalar Positives
- VLIW Positives
HW support for More ILP

- **Speculation:** allow an instruction to issue that is dependent on branch predicted to be taken *without* any consequences (including exceptions) if branch is not actually taken (“HW undo”)
- Often combined with dynamic scheduling
- Tomasulo: separate *speculative* bypassing of results from real bypassing of results
  - When instruction no longer speculative, write results (*instruction commit*)
  - execute out-of-order but commit in order

Hardware Speculative Execution

- Need HW buffer for results of uncommitted instructions: **reorder buffer**
  - Reorder buffer can be operand source
  - Once operand commits, result is found in register
  - 3 fields: instr. type, destination, value
  - Use reorder buffer number instead of reservation station
  - Instructions commit in order
  - As a result, its easy to undo speculated instructions on mispredicted branches or on exceptions

Four Steps of Speculative Tomasulo Algorithm

1. **Issue**—get instruction from FP Op Queue
   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination. Operands may be read from register file or reorder buffer.

2. **Execution**—operate on operands (EX)
   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute

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

4. **Commit**—update register with reorder result
   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer.
Speculative Execution

- The re-order buffer and in-order commit allow us to flush the speculative instructions from the machine when a misprediction is discovered.
- ROB is another possible source of operands.
- ROB can provide precise exceptions in an out-of-order machine.
- ROB allows us to ignore exceptions on speculative code.

- Compiler speculation vs. hardware speculation?

Compiler support for ILP: Software Pipelining

- Observation: if iterations from loops are independent, then can get ILP by taking instructions from different iterations.
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop.

SW Pipelining Example

<table>
<thead>
<tr>
<th>Unrolled 3 times</th>
<th>Software Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 LD F0,0(R1)</td>
<td>1 SD 0(R1),F4; Stores M[i]</td>
</tr>
<tr>
<td>2 ADDD F4,F0,F2</td>
<td>2 ADDD F4,F0,F2; Adds to M[i-1]</td>
</tr>
<tr>
<td>3 SD 0(R1),F4</td>
<td>3 LD F0,-16(R1); loads M[i-2]</td>
</tr>
<tr>
<td>4 LD F6,-8(R1)</td>
<td>4 SUBI R1,R1,#8</td>
</tr>
<tr>
<td>5 ADDD F8,F6,F2</td>
<td>5 BNEZ R1,LOOP</td>
</tr>
<tr>
<td>6 SD -8(R1),F8</td>
<td>6 SD 0(R1),F4</td>
</tr>
<tr>
<td>7 LD F10,-16(R1)</td>
<td>7 ADDD F4,F0,F2</td>
</tr>
<tr>
<td>8 ADDD F12,F10,F2</td>
<td>8 SD -8(R1),F4</td>
</tr>
<tr>
<td>9 SD -16(R1),F12</td>
<td>9 BNEZ R1,LOOP</td>
</tr>
<tr>
<td>10 SUBI R1,R1,#24</td>
<td>10 SUBI R1,R1,#24</td>
</tr>
<tr>
<td>11 BNEZ R1,LOOP</td>
<td>11 BNEZ R1,LOOP</td>
</tr>
</tbody>
</table>

Compiler Support for ILP: Trace Scheduling

- Creates long basic blocks by finding long paths in the code.
Trace Scheduling

- Parallelism across IF branches vs. LOOP branches
- Two steps:
  - Trace Selection
    - Find likely sequence of basic blocks (trace) of (statically predicted) long sequence of straight-line code
  - Trace Compaction
    - Squeeze trace into few VLIW instructions
    - Need bookkeeping code in case prediction is wrong

HW support for More ILP

- Avoid branch prediction by turning branches into conditionally executed instructions: (aka predicated instructions)
  
  if (x) then A = B op C else NOP
  
  - If false, then neither store result or cause exception
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr, IA64 can predicate any instruction (even have multiple predicates)

<table>
<thead>
<tr>
<th>Instruction issues per cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc                         49</td>
</tr>
<tr>
<td>espresso                    75</td>
</tr>
<tr>
<td>li                          119</td>
</tr>
<tr>
<td>fpppp                       35</td>
</tr>
<tr>
<td>doduc                      14</td>
</tr>
<tr>
<td>tomcatv                    80</td>
</tr>
</tbody>
</table>

Predicated Execution

- Drawbacks to conditional instructions
  - Still takes a clock & alu even if “annulled”
  - Stall if condition evaluated late
  - Complex conditions reduce effectiveness; condition becomes known late in pipeline
  - Requires more operands! Typically only available as conditional move.

- Advantages
  - Eliminate prediction, misprediction
  - Longer basic blocks, ...

ILP in real code

- Based on all kinds of ideal assumptions. Further limited by:
  - Realistic branch prediction
  - Finite renaming registers
  - Imperfect alias analysis for memory operations

FIGURE 4.40 The effect of window size shown by each application by plotting the average number of instruction issues per clock cycle.
PowerPC 620

- speculative, out-of-order, superscalar processor
- fetches 4, issues 4, completes 4, 6 independent functional units
- Reservation stations, reorder buffer, register renaming in the register file

PowerPC 620 Pipeline Structure

- Fetch
- Decode
- Issue
- Execute
- Commit

PowerPC 620 Performance

- FIGURE 4.52 The average number of instructions that the fetch unit can provide to the issue unit varies between 3.2 and 4, with an average of 3.4 for the integer benchmarks and 3.8 for the FP benchmarks.

- FIGURE 4.54 The IPC throughput rate for the issue stage is arrived at by subtracting stalls that arise in issue from the IPC rate sustained by the fetch stage.
PowerPC 620 Performance

![Graph showing IPC at the issue stage](image)

**FIGURE 4.57** The breakdown of the ideal IPC of 4.0 into its components.

### ILP Summary

- Parallelism is absolutely critical to modern computer system performance, but at a very fine level.
- Mechanisms that create, or expose parallelism: loop unrolling, software pipelining, code motion
- Mechanisms that allow the machine to exploit ILP: pipelining, superscalar, dynamic scheduling, speculative execution