HW support for More ILP

- **Speculation**: allow an instruction to issue that is dependent on branch without any consequences (including exceptions) if branch is predicted incorrectly (“HW undo”)
- Often combined with ____________
- Tomasulo: separate _______ bypassing of results from ________ bypassing of results
  - When instruction no longer speculative, write results (*instruction commit* or *instruction retire*)
  - execute out-of-order but commit in order
  - Requires some kind of intermediate storage

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 as “name” of result
  - 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.

Tomasulo – cycle 0

Loop:
- ADDD: F4, F2, F0
- MULD: F8, F4, F2
- ADDD: F6, F8, F6
- SUBD: F8, F2, F0
- SUBI: ...
- BNEZ: ...

Instruction Queue

<table>
<thead>
<tr>
<th>ROB</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

FP adders

FP mul’s
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 __________ in an out-of-order machine
- ROB allows us to ______ exceptions on speculative code.
- Compiler speculation vs. hardware speculation?

Now what?

- CPI = 1.0 + BSPI + FPSPi + LdSPI

Multiple Instruction Issue

The insts go marching two by two, hurrah, hurrah...
Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar
  - variable number of instructions issued each cycle
  - parallelism detected in hardware
- Very Long Instruction Words (VLIW)
  - fixed number of instructions issued each cycle
  - parallelism scheduled by the compiler
  - IA-64 (Itanium)

Superscalar MIPS Implications

- More ports for FP registers to do FP load & FP op in a pair
- 1 cycle load delay expands to 3 instructions in SS
  - instruction in right half can’t use it, nor instructions in next slot
- Branch delay also expands to 3

Let’s revisit our favorite loop….

Unrolled Loop that Minimizes Stalls for Scalar

```
1 Loop:  LD  F0, 0(R1)  LD to ADDD: 2 Cycle
        LD  F6, -8(R1)
        LD  F10, -16(R1)
        ADDD F4, F0, F2
        ADDD F8, F6, F2
        ADDD F12, F10, F2
        ADDD F16, F14, F2
        SD  0(R1), F4
        SD  -8(R1), F8
        SD  -16(R1), F12
        SUBI R1, R1, #32
        BNEZ R1, LOOP
        SD  8(R1), F16  ; 8 - 32 = -24
```

14 clock cycles, or 3.5 per iteration
**Loop Unrolling in Superscalar**

- Integer instruction  | FP instruction  | Clock cycle
- LD **(F0,0(R1))**  | ADDD F4,F0,F2  | 1
- LD **F6,-8(R1)**   | ADDD F8,F6,F2  | 2
- LD **F10,-16(R1)** | ADDD F12,F10,F2| 3
- LD **F14,-24(R1)** | ADDD F16,F16,F2| 4
- LD **F18,-32(R1)** | ADDD F20,F18,F2| 5
- SD 0(R1),F8       | ADDD F20,F18,F2| 6
- SD **-8(R1),F8**  | ADDD F20,F18,F2| 7
- SD **-16(R1),F12**| ADDD F20,F18,F2| 8
- SD **-24(R1),F16**| ADDD F20,F18,F2| 9
- SUBI R1,R1,#40    | ADDD F20,F18,F2| 10
- BNEZ R1,LOOP     | ADDD F20,F18,F2| 11
- SD **-32(R1),F20**| ADDD F20,F18,F2| 12

- Unrolled 5 times to avoid delays
- 12 clocks, or 2.4 clocks per iteration

**Dynamic Scheduling and Superscalar**

- Dependencies stop instruction issue in In-order SS
- Code compiled for scalar version will run poorly on SS
  - May want code to vary depending on how superscalar
- Simple approach: Combine dynamic scheduling (e.g., Tomasulo) with the ability to fetch and issue multiple instructions simultaneously
  - simplified if we don’t issue instructions that read the same register file in the same cycle.
  - requires multiple CDBs
  - can complicate updating of register bookkeeping if instructions are dependent, but can be done

**Superscalar Dynamic Issue**

- Issues/complications?

**Performance of Dynamic SS**

<table>
<thead>
<tr>
<th>Iteration Instructions</th>
<th>Issues</th>
<th>Executes</th>
<th>Writes result</th>
</tr>
</thead>
<tbody>
<tr>
<td>no.</td>
<td>clock-cycle number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LD <strong>F0,0(R1)</strong></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>ADDD F4,F0,F2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SD 0(R1),F4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SUBI R1,R1,#8</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BNEZ R1,LOOP</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LD <strong>F0,0(R1)</strong></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADDD F4,F0,F2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SD 0(R1),F4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SUBI R1,R1,#8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BNEZ R1,LOOP</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

- 4 clocks per iteration (bottleneck?)

Branches, Decrements still take 1 clock cycle

- 4 cycle “latency” (in execute 3, write result 4th, use 5th)
- Ld 3 cycle “latency” (in execute 2, write result 3rd, use 4th)
Limits of Superscalar

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - 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

Superscalar Key Points

- Only way to get CPI < 1 is multiple instruction issue
- SS requires duplicated hardware, more dependence checking
- Without duplication of functional units, will see limited improvement
- SS combined with dynamic scheduling can be powerful

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 | multd f6, f4, f2 | lw r2, 0(r7) | sub r8, r6, r1 | beq z r9, label
sub r1, r5, r1 | add r8, r6, r1 | sw r5, 8(r7) | nop | beq r2, label

Loop Unrolling in VLIW

- 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