Lecture 19: Instruction Level Parallelism
-- Dynamic Scheduling, Multiple Issue, and Speculation

CSE 564 Computer Architecture Summer 2017

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
Topics for Instruction Level Parallelism

- **ILP Introduction, Compiler Techniques and Branch Prediction**
  - 3.1, 3.2, 3.3

- **Dynamic Scheduling (OOO)**
  - 3.4, 3.5 and C.5, C.6 and C.7 (FP pipeline and scoreboard)

- **Hardware Speculation and Static Superscalar/VLIW**
  - 3.6, 3.7

- **Dynamic Scheduling, Multiple Issue and Speculation**
  - 3.8, 3.9, 3.13

- **ILP Limitations and SMT**
  - 3.10, 3.11, 3.12
Acknowledge and Copyright

- Slides adapted from
  - UC Berkeley course “Computer Science 252: Graduate Computer Architecture” of David E. Culler Copyright(C) 2005 UCB
  - UC Berkeley course Computer Science 252, Graduate Computer Architecture Spring 2012 of John Kubiatowicz Copyright(C) 2012 UCB
  - Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley
  - Arvind (MIT), Krste Asanovic (MIT/UCB), Joel Emer (Intel/MIT), James Hoe (CMU), John Kubiatowicz (UCB), and David Patterson (UCB)

- https://passlab.github.io/CSE564/copyrightack.html
REVIEW
Not Every Stage Takes only one Cycle

- **FP EXE Stage**
  - Multi-cycle Add/Mul
  - Nonpipelined for DIV

- **MEM Stage**

---

Figure C.41 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating
Issues of Multi-Cycle in Some Stages

- The divide unit is not fully pipelined
  - structural hazards can occur
    » need to be detected and stall incurred.

- The instructions have varying running times
  - the number of register writes required in a cycle can be $> 1$

- Instructions no longer reach WB in order
  - Write after write (WAW) hazards are possible
    » Note that write after read (WAR) hazards are not possible, since the register reads always occur in ID.

- Instructions can complete in a different order than they were issued (out-of-order complete)
  - causing problems with exceptions

- Longer latency of operations
  - stalls for RAW hazards will be more frequent.
Dynamic Scheduling and Speculation

- **ILP Maximized (a restricted data-flow)**
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
  - In-order commit

- **Data Hazards**
  - Input operands-driven dynamic scheduling for RAW hazard
  - Register renaming for handling WAR and WAW hazards

- **Control Hazards (Branching, Precision Exception)**
  - Branch prediction and in-order commit (speculation)
  - Branch prediction without speculation
    » Cannot do out-of-order execution/complete for branch

- **Implementation: Tomasulo**
  - Reservation stations and Reorder buffer
  - Other solutions as well (scoreboard, history table)
MULTIPLE ISSUE VIA VLIW/STATIC SUPERSCALAR
Multiple Issue

- "Flynn bottleneck"
  - single issue performance limit is CPI = IPC = 1
  - hazards + overhead \(\Rightarrow\) CPI \(\geq\) 1 (IPC \(\leq\) 1)
  - diminishing returns from superpipelining [Hrishikesh paper!]

- Solution: issue multiple instructions per cycle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst0</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst2</td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>inst3</td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
</tbody>
</table>

- 1st superscalar: IBM America \(\rightarrow\) RS/6000 \(\rightarrow\) POWER1
VLIW: Very Large Instruction Word

- Each “instruction” has explicit coding for multiple operations
  - In IA-64, grouping called a “packet”
  - In Transmeta, grouping called a “molecule” (with “atoms” as ops)

- 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., 1 integer operation/branch, 2 FP ops, 2 Memory refs
    » 16 to 24 bits per field => 5*16 or 80 bits to 5*24 or 120 bits wide
  - Need compiling technique that schedules across several branches
Recall: Unrolled Loop that Minimizes Stalls for Scalar

for \((i=999; i>=0; i=i-1)\)

\[ x[i] = x[i] + s; \]

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the average number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0, since the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0.

14 clock cycles, or 3.5 per iteration
Loop Unrolling in VLIW

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

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP operation 2</th>
<th>Integer operation/branch</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>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td></td>
<td></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>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td></td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></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>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td>S.D F16,-24(R1)</td>
<td></td>
<td>DADDUI R1,R1,#-56</td>
<td></td>
</tr>
<tr>
<td>S.D F20,24(R1)</td>
<td>S.D F24,16(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F28,8(R1)</td>
<td></td>
<td></td>
<td>BNE R1,R2,Loop</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.16 VLIW instructions that cycles assuming no branch delay; normal operations in 9 clock cycles, or 2.5 operations, is about 60%. To achieve this this loop. The VLIW code sequence also MIPS processor can use as few as two F

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>

12
Very Important Terms

- **Dynamic Scheduling** → Out-of-order Execution
- **Speculation** → In-order Commit
- **Superscalar** → Multiple Issue

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Goals</th>
<th>Implementation</th>
<th>Addressing</th>
<th>Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Scheduling</td>
<td>Out-of-order execution</td>
<td>Reservation Stations, Load/Store Buffer and CDB</td>
<td>Data hazards (RAW, WAW, WAR)</td>
<td>Register renaming</td>
</tr>
<tr>
<td>Speculation</td>
<td>In-order commit</td>
<td>Branch Prediction (BHT/BTB) and Reorder Buffer</td>
<td>Control hazards (branch, func, exception)</td>
<td>Prediction and misprediction recovery</td>
</tr>
<tr>
<td>Superscalar/VLIW</td>
<td>Multiple issue</td>
<td>Software and Hardware</td>
<td>To Increase CPI</td>
<td>By compiler or hardware</td>
</tr>
</tbody>
</table>
DYNAMIC SCHEDULING, MULTIPLE ISSUE (DYNAMIC SUPERSCALAR), AND SPECULATION
Example

Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

```assembly
Loop:   LD   R2,0(R1) ;R2=array element
       DADDIU R2,R2,#1 ;increment R2
       SD    R2,0(R1) ;store result
       DADDIU R1,R1,#8 ;increment pointer
       BNE   R2,R3,LOOP ;branch if not last element
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.
With Speculation

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Instructions</th>
<th>Issues at clock number</th>
<th>Execute at clock number</th>
<th>at clock number</th>
<th>clock number</th>
<th>at clock number</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD R2,0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>First issue</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R2,R2,#1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>1</td>
<td>SD R2,0(R1)</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R1,R1,#8</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>Commit in order</td>
</tr>
<tr>
<td>1</td>
<td>BNE R2,R3,LOOP</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>LD R2,0(R1)</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>No execute delay</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R2,R2,#1</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
<td>Wait for LW</td>
</tr>
<tr>
<td>2</td>
<td>SD R2,0(R1)</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R1,R1,#8</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td></td>
<td>Commit in order</td>
</tr>
<tr>
<td>2</td>
<td>BNE R2,R3,LOOP</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>LD R2,0(R1)</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>Earliest possible</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R2,R2,#1</td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td>Wait for LW</td>
</tr>
<tr>
<td>3</td>
<td>SD R2,0(R1)</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R1,R1,#8</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>14</td>
<td></td>
<td>Executes earlier</td>
</tr>
<tr>
<td>3</td>
<td>BNE R2,R3,LOOP</td>
<td>9</td>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
</tbody>
</table>

Figure 3.20 The time of issue, execution, and writing result for a dual-issue version of our pipeline with speculation. Note that the LD following the BNE can start execution early because it is speculative.
ADVANCED TECHNIQUES FOR INSTRUCTION DELIVERY AND SPECULATION

1. Advanced Branch Prediction
2. Explicit Register Renaming
3. Others that are important but not covered: Load/store speculation, value predication, correlate branch prediction, tournament predictor, trace cache
4. Put all together on ARM Cortex-A8 and Intel Core i7
ILP LIMITATIONS
Instruction queue/window

- Buffer to hold decoded or ready-to-issue instructions

Figure 3.17 The basic organization of a multiple issue processor with speculation. In this case, the organization could allow a FP multiply, FP add, integer, and load/store to all issues simultaneously (assuming one issue per clock per functional unit). Note that several datapaths must be widened to support multiple issues: the CDB, the operand buses, and, critically, the instruction issue logic, which is not shown in this figure. The last is a difficult problem, as we discuss in the text.
Limits to ILP

Initial HW Model here; compiler close to magic.
Assumptions for ideal/perfect machine to start:

1. *Register renaming* – infinite virtual registers
   ⇒ all register WAW & WAR hazards are avoided
2. *Branch prediction* – perfect; no mispredictions
3. *Jump prediction* – all jumps perfectly predicted
   (returns, case statements)
2 & 3 ⇒ no control dependencies; perfect speculation
   & an unbounded buffer of instructions available
4. *Memory-address alias analysis* – addresses known
   & a load can be moved before a store provided
   addresses not equal; 1&4 eliminates all but RAW

Also: perfect caches; 1 cycle latency for all instructions
   (FP *,/); unlimited instructions issued/clock cycle;
# Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Issued per clock</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>Perfect</td>
<td>2% to 6% misprediction (Tournament Branch Predictor)</td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td>Memory Alias Analysis</td>
<td>Perfect</td>
<td>??</td>
</tr>
</tbody>
</table>
Upper Limit to ILP: Ideal Machine

Instructions Per Clock

FP: 75 - 150

Integer: 18 - 60

<table>
<thead>
<tr>
<th>Programs</th>
<th>Instructions Per Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>54.8</td>
</tr>
<tr>
<td>espresso</td>
<td>62.6</td>
</tr>
<tr>
<td>li</td>
<td>17.9</td>
</tr>
<tr>
<td>fpppp</td>
<td>75.2</td>
</tr>
<tr>
<td>doducd</td>
<td>118.7</td>
</tr>
<tr>
<td>tomcatv</td>
<td>150.1</td>
</tr>
</tbody>
</table>
# Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Issued per clock</td>
<td>Infinite</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td>Infinite, 2K, 512, 128, 32</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>Infinite</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>Perfect</td>
<td>Perfect</td>
<td>2% to 6% misprediction (Tournament Branch Predictor)</td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td>Memory Alias</td>
<td>Perfect</td>
<td>Perfect</td>
<td>??</td>
</tr>
</tbody>
</table>
More Realistic HW: Window Impact

Instruction window changed from Infinite window 2048, 512, 128, 32

FP: 9 - 150

Integer: 8 - 63
Realistic HW: Window Impact

Perfect disambiguation (HW), 1K Selective Prediction, 16 entry return, 64 registers, issue as many as window

Program Issue per Cycle

FP: 8 - 45

Integer: 6 - 12

Program

gcc   expresso   li   fpppp   doducd   tomcatv

Infinite 25 6  12 8  64  32  16  8  4

IPC

25
## Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instructions Issued per clock</strong></td>
<td>64</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td><strong>Instruction Window Size</strong></td>
<td>2048</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td><strong>Renaming Registers</strong></td>
<td>Infinite</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td><strong>Branch Prediction</strong></td>
<td>Perfect vs. 8K Tournament vs. 512 2-bit vs. profile vs. none</td>
<td>Perfect</td>
<td>2% to 6% misprediction (Tournament Branch Predictor)</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td><strong>Memory Alias</strong></td>
<td>Perfect</td>
<td>Perfect</td>
<td>??</td>
</tr>
</tbody>
</table>
More Realistic HW: Branch Impact

Change from Infinite window to examine to 2048 and maximum issue of 64 instructions per clock cycle

FP: 15 - 45

Integer: 6 - 12

IPC

Instruction issues per cycle

Program

gcc  espresso  li  fpppp  doducd  tomcatv

Perfect  Selective predictor  Standard 2-bit  Static  None
Misprediction Rates

- Misprediction Rates for different programs:
  - tomcatv: 1%
  - doduc: 5%
  - fpppp: 16%
  - li: 18%
  - espresso: 14%
  - gcc: 30%

Legend:
- Red: Profile-based
- Green: 2-bit counter
- Blue: Tournament
### Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Issued per clock</td>
<td>64</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td>2048</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>Infinite v. 256, 128, 64, 32, none</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>8K 2-bit</td>
<td>Perfect</td>
<td>Tournament Branch Predictor</td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td>Memory Alias</td>
<td>Perfect</td>
<td>Perfect</td>
<td>Perfect</td>
</tr>
</tbody>
</table>


More Realistic HW:
Renaming Register Impact (N int + N fp)

Change 2048 instr window, 64 instr issue, 8K 2 level Prediction

FP: 11 - 45

Integer: 5 - 15
## Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Issued per clock</td>
<td>64</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td>2048</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>256 Int + 256 FP</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>8K 2-bit</td>
<td>Perfect</td>
<td>Tournament</td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td>Memory Alias</td>
<td>Perfect v. Stack v. Inspect v. none</td>
<td>Perfect</td>
<td>Perfect</td>
</tr>
</tbody>
</table>
More Realistic HW:
Memory Address Alias Impact

Change 2048 instr window, 64 instr issue, 8K 2 level Prediction, 256 renaming registers

Program Instruction issues per cycle

<table>
<thead>
<tr>
<th>Program</th>
<th>IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>10</td>
</tr>
<tr>
<td>espresso</td>
<td>15</td>
</tr>
<tr>
<td>li</td>
<td>12</td>
</tr>
<tr>
<td>fpppp</td>
<td>49</td>
</tr>
<tr>
<td>doducd</td>
<td>16</td>
</tr>
<tr>
<td>tomcatv</td>
<td>45</td>
</tr>
</tbody>
</table>

FP: 4 – 45 (Fortran, no heap)

Integer: 4 - 9

Perfect Gobal/stack Perfect Inspection None
## Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions issued per clock</td>
<td>64 (no restrictions)</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td>Infinite vs. 256, 128, 64, 32</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>64 Int + 64 FP</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>1K 2-bit</td>
<td>Perfect</td>
<td>Tournament</td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td>Memory Alias</td>
<td>HW disambiguation</td>
<td>Perfect</td>
<td>Perfect</td>
</tr>
</tbody>
</table>
How to Exceed ILP Limits of this study?

- These are not laws of physics; just practical limits for today, and perhaps overcome via research
- Compiler and ISA advances could change results
- WAR and WAW hazards through memory: eliminated WAW and WAR hazards through register renaming, but not in memory usage
  - Can get conflicts via allocation of stack frames as a called procedure reuses the memory addresses of a previous frame on the stack
HW v. SW to increase ILP

- Memory disambiguation: HW best
- Speculation:
  - HW best when dynamic branch prediction better than compile time prediction
  - Exceptions easier for HW
  - HW doesn’t need bookkeeping code or compensation code
  - Very complicated to get right
- Scheduling: SW can look ahead to schedule better
- Compiler independence: does not require new compiler, recompilation to run well
Take-away Message

- There is limitation of ILP we can achieve regardless of the # of transistors we can use
  - Moore’s law → performance limitation

- Guideline and experiment for improving ILP, a.k. CPI
  - Application dependent

- Per-core performance so far
THREAD-LEVEL PARALLELISM AND SIMULTANEOUS MULTITHREADING (SMT)
Pipeline Hazards in ILP

- Each instruction may depend on the next

```
LW r1, 0(r2)
LW r5, 12(r1)
ADDI r5, r5, #12
SW 12(r1), r5
```

What is usually done to cope with this?

- interlocks (slow)
- or bypassing (needs hardware, doesn’t help all hazards)
Multithreading

- Difficult to continue to extract instruction-level parallelism (ILP) from a single sequential thread of control
- Many workloads can make use of thread-level parallelism (TLP)
  - TLP from multiprogramming (run independent sequential jobs)
  - TLP from multithreaded applications (run one job faster using parallel threads)
- Multithreading uses TLP to improve utilization of a single processor
Multithread Program in OpenMP

```
$ gcc -fopenmp hello.c

$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
Hello World

$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World
Hello World
Hello World
Hello World

#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        printf("Hello World\n");
    } // End of parallel region
    return(0);
}
```
## Typical OpenMP Parallel Program

<table>
<thead>
<tr>
<th>Sequential code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(i=0;i&lt;N;i++) { a[i] = a[i] + b[i]; }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OpenMP <code>parallel</code> region</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp parallel shared (a, b)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>int id, i, Nthrds, istart, iend;</td>
</tr>
<tr>
<td>id = omp_get_thread_num();</td>
</tr>
<tr>
<td>Nthrds = omp_get_num_threads();</td>
</tr>
<tr>
<td>istart = id * N / Nthrds;</td>
</tr>
<tr>
<td>iend = (id+1) * N / Nthrds;</td>
</tr>
<tr>
<td>for(i=istart;i&lt;iend;i++) { a[i] = a[i] + b[i]; }</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OpenMP <code>parallel</code> region and a worksharing <code>for</code> construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp parallel shared (a, b) private (i)</td>
</tr>
<tr>
<td>#pragma omp for schedule(static)</td>
</tr>
<tr>
<td>for(i=0;i&lt;N;i++) { a[i] = a[i] + b[i]; }</td>
</tr>
</tbody>
</table>
Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

-- One way is to interleave execution of instructions from different program threads on same pipeline

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe

Prior instruction in a thread always completes write-back before next instruction in same thread reads register file
CDC 6600 Peripheral Processors  
(Cray, 1964)

- First multithreaded hardware
- 10 “virtual” I/O processors
- Fixed interleave on simple pipeline
- Pipeline has 100ns cycle time
- Each virtual processor executes one instruction every 1000ns
- Accumulator-based instruction set to reduce processor state
Performance beyond single thread ILP

- There can be much higher natural parallelism in some applications
  - e.g., Database or Scientific codes
  - Explicit Thread Level Parallelism or Data Level Parallelism

- **Thread**: instruction stream with own PC and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute

- **Thread Level Parallelism (TLP)**:
  - Exploit the parallelism inherent between threads to improve performance

- **Data Level Parallelism (DLP)**:
  - Perform identical operations on data, and lots of data
One approach to exploiting threads: Multithreading (TLP within processor)

- Multithreading: multiple threads to share the functional units of 1 processor via overlapping
  - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - HW for fast thread switch; much faster than full process switch $\approx 100s$ to $1000s$ of clocks

- When switch?
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)
## Multithreaded Categories

### Time (processor cycle)

<table>
<thead>
<tr>
<th>Superscalar</th>
<th>Fine-Grained</th>
<th>Coarse-Grained</th>
<th>Multiprocessing</th>
<th>Simultaneous Multithreading</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Superscalar Details" /></td>
<td><img src="#" alt="Fine-Grained Details" /></td>
<td><img src="#" alt="Coarse-Grained Details" /></td>
<td><img src="#" alt="Multiprocessing Details" /></td>
<td><img src="#" alt="Simultaneous Multithreading Details" /></td>
</tr>
</tbody>
</table>

### Thread Colors
- **Thread 1**: Light Blue
- **Thread 2**: Red
- **Thread 3**: Yellow
- **Thread 4**: Green
- **Thread 5**: Pink
- **Idle slot**: White

---

**Legend**
- **Thread 1**: Light Blue
- **Thread 2**: Red
- **Thread 3**: Yellow
- **Thread 4**: Green
- **Thread 5**: Pink
- **Idle slot**: White
Course-Grained Multithreading

- Switches threads only on costly stalls, such as L2 cache misses

- Advantages
  - Relieves need to have very fast thread-switching
  - Doesn’t slow down thread, since instructions from other threads issued only when the thread encounters a costly stall

- Disadvantage is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
  - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
  - New thread must fill pipeline before instructions can complete

- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time

- Used in IBM AS/400, Sparcle (for Alewife)
Fine-Grained Multithreading

- Switches between threads on each instruction, causing the execution of multiples threads to be interleaved
  - Usually done in a round-robin fashion, skipping any stalled threads
  - CPU must be able to switch threads every clock

- Advantage:
  - can hide both short and long stalls, since instructions from other threads executed when one thread stalls

- Disadvantage:
  - slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads

- Used on Oracle SPARC processor (Niagra from Sun), several research multiprocessors, Tera
Simultaneous Multithreading (SMT): Do both ILP and TLP

- TLP and ILP exploit two different kinds of parallel structure in a program.
- Could a processor oriented at ILP to exploit TLP?
  - Functional units are often idle in data path designed for ILP because of either stalls or dependences in the code.
- Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?
Simultaneous Multi-threading ...

<table>
<thead>
<tr>
<th>Cycle</th>
<th>M</th>
<th>M</th>
<th>FX</th>
<th>FX</th>
<th>FP</th>
<th>FP</th>
<th>BR</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle</th>
<th>M</th>
<th>M</th>
<th>FX</th>
<th>FX</th>
<th>FP</th>
<th>FP</th>
<th>BR</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes
Choosing Policy

- Among four threads, from which do we fetch?
  - Fetch from thread with the least instructions in flight.
Simultaneous Multithreading Details

- Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
  - Large set of virtual registers that can be used to hold the register sets of independent threads
  - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads
  - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW

- Just adding a per thread renaming table and keeping separate PCs
  - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

Source: Micrprocessor Report, December 6, 1999
“Compaq Chooses SMT for Alpha”
Design Challenges in SMT

- Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance?
  - A preferred thread approach sacrifices neither throughput nor single-thread performance?
  - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls

- Larger register file needed to hold multiple contexts

- Clock cycle time, especially in:
  - Instruction issue - more candidate instructions need to be considered
  - Instruction completion - choosing which instructions to commit may be challenging

- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance
Simple Multithreaded Pipeline

- Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage
- Appears to software (including OS) as multiple, albeit slower, CPUs
Multithreading Costs

- Each thread requires its own user state
  - PC
  - GPRs

- Also, needs its own system state
  - Virtual-memory page-table-base register
  - Exception-handling registers

- Other overheads:
  - Additional cache/TLB conflicts from competing threads
  - (or add larger cache/TLB capacity)
  - More OS overhead to schedule more threads (where do all these threads come from?)
For most apps, most execution units lie idle in an OoO superscalar.

For an 8-way superscalar.

“Processor busy” are the actual used issue slots.

O-o-O Simultaneous Multithreading
[Tullsen, Eggers, Emer, Levy, Stamm, Lo, DEC/UW, 1996]

- Add multiple contexts and fetch engines and allow instructions fetched from different threads to issue simultaneously
- Utilize wide out-of-order superscalar processor issue queue to find instructions to issue from multiple threads
- OOO instruction window already has most of the circuitry required to schedule from multiple threads
- Any single thread can utilize whole machine
Power 4

Single-threaded predecessor to Power 5. 8 execution units in out-of-order engine, each may issue an instruction each cycle.
Power 4

- Branch redirects
- Instruction fetch
- 2 fetch (PC)
- 2 initial decodes
- Instruction crack and group formation
- Out-of-order processing

Power 5

- Branch redirects
- Instruction fetch
- 2 fetch (PC)
- 2 initial decodes
- Out-of-order processing

2 commits
(architected register sets)
Power 5 data flow...

Why only 2 threads? With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck
Changes in Power 5 to support SMT

- Increased associativity of L1 instruction cache and the instruction address translation buffers
- Added per thread load and store queues
- Increased size of the L2 (1.92 vs. 1.44 MB) and L3 caches
- Added separate instruction prefetch and buffering per thread
- Increased the number of virtual registers from 152 to 240
- Increased the size of several issue queues
- The Power5 core is about 24% larger than the Power4 core because of the addition of SMT support
First commercial SMT design (2-way SMT)
  - Hyperthreading == SMT

Logical processors share nearly all resources of the physical processor
  - Caches, execution units, branch predictors

Die area overhead of hyperthreading \( \sim 5\% \)

When one logical processor is stalled, the other can make progress
  - No logical processor can use all entries in queues when two threads are active

Processor running only one active software thread runs at approximately same speed with or without hyperthreading

Hyperthreading dropped on OoO P6 based follow-ons to Pentium-4 (Pentium-M, Core Duo, Core 2 Duo), until revived with Nehalem generation machines in 2008.

Intel Atom (in-order x86 core) has two-way vertical multithreading
Initial Performance of SMT

- Pentium 4 Extreme SMT yields 1.01 speedup for SPECint_rate benchmark and 1.07 for SPECfp_rate
  - Pentium 4 is dual threaded SMT
  - SPECRate requires that each SPEC benchmark be run against a vendor-selected number of copies of the same benchmark

- Running on Pentium 4 each of 26 SPEC benchmarks paired with every other (26^2 runs) speed-ups from 0.90 to 1.58; average was 1.20

- Power 5, 8-processor server 1.23 faster for SPECint_rate with SMT, 1.16 faster for SPECfp_rate

- Power 5 running 2 copies of each app speedup between 0.89 and 1.41
  - Most gained some
  - Fl.Pt. apps had most cache conflicts and least gains
Intel i7 Performance

- 2-thread SMT
End of Chapter 3