Outline

• Announcements
  – Strongly encourage you to stop working on Assignment 1
    • Contact me to get the reference code for Assignment 2 (due next week)
  – Final exam on December 11th
  – Next lecture (October 23rd): Professor Ben Goldberg
    • Intel IA 64 architecture, EPIC, and compiling for it

• Last lecture: Tomasulo’s algorithm
• Further reducing pipeline stalls
  – Conditional instructions
  – Hardware speculation
• Multiple-issue processors (achieving IPC > 1)

[Hennessy/Patterson CA:AQA (3rd Edition): Chapter 3]
Tomasulo’s Algorithm and Precise Interrupts

- **Precise interrupts**
  - Instructions before the faulting instruction are allowed to complete
  - Instructions after the faulting instruction can be restarted from scratch - should not change processor state before the exception is taken.
  - Example: Correctly handling page faults
    
    ```
    ADD R1, R2, R3
    MUL R1, R2, R4
    LOAD R5, 200(R6)
    ADD R1, R5, R7
    MUL R1, R1, R6
    ```
  - Standard Tomasulo’s algorithm rewrites destination register if dependencies for prior instructions can be handled through tag forwarding
    - See Loop example from previous lecture

Extended Tomasulo’s Algorithm

- Force instructions to “commit” in order
  - i.e., processor register state is seen in order, supporting precise exceptions
- Typically implemented using a ReOrder Buffer
  - Holds results till they are ready to be written to registers
  - Instructions can use ROB values
- Same support allows hardware speculation
  - Later in the lecture
Alternative Implementations of Tomasulo-like Ideas


• Tomasulo’s algorithm requires comparison logic for tag-matching
  – Compare tag associated with register to tag identifying RS
  – Needs to be done per register
    • Not practical: model architecture in paper required 144 such units

• Proposal 1: Separate out tag matching hardware into a Tag Unit
  – Can have size independent of the register set size
    • only need tags for active registers
  – Responsible for result bypassing (tag forwarding), updating registers

• Proposal 2: Merge RS and TU into a combined unit
  – RS’s shared amongst functional units: Advantages? Disadvantages?
  – When the merged unit is implemented as a queue: Register Update Unit
    • sim-outoforder uses this model of dynamic scheduling

Modified Architecture with RUU

• RUU serves the same role as a ROB for precise exceptions
HW Support for Reducing Stalls

- Dynamic scheduling schemes wait for branch instructions to be resolved before executing a (dependent) instruction
  - These instructions can be issued, but are not executed

- Branch prediction schemes help minimize wasted issue effort
  - But do not by themselves accelerate execution of dependent instructions
  - Moreover, there will always be mispredicted branches

Can we do better?
Yes, with more hardware support

- Conditional instructions
  - Execute the (dependent) instruction, but does not have any effect if branch ends up not being taken

- Hardware speculation
  - Execute the (dependent) instruction, but state does not get “committed”

Hardware Support (1): Conditional Instructions

- Avoid need for branch prediction by turning branches into conditionally executed instructions
  - Trading off branch mispredict penalty for unused instructions
  - Bigger advantage: Allows compilers the freedom to reorder instructions

- The code
  
  ```
  if (R1 == 0) R2 = R3;
  ```

  normally would be implemented as

  ```
  BNEZ R1, L
  ADD     R2, R3, R0
  ```

- With conditional move instructions, this would be implemented as

  ```
  CMOVZ R2, R3, R1
  ```

  - If R1 == 0, then the instruction has no effect (and does not cause any exceptions)
Conditional Instructions

- **Advantages**
  - Can eliminate simple branches
  - Can reduce code size

- **Drawbacks**
  - Still takes a clock even if condition is false
  - Needs the condition to be evaluated early to be useful
  - Could result in higher CPI or lower clock rate
  - Could make control and datapath design complex

- **Conditional instructions useful for simple conditions and instructions**
  - ISAs in Alpha, MIPS, PowerPC, SPARC have conditional move
  - HP PA: any reg-reg instruction can nullify the following instruction, making it conditional
  - IA-64: 64 1-bit predicate register fields allow any instruction to be executed conditionally

Hardware Support (2): Speculative Execution

- **Main idea:** allow execution of an instruction dependent on branch predicted to be taken without any consequences (including exceptions) if branch is not actually taken
  - Hardware needs to undo the instruction - hard to do if there are exceptions

- **Need to consider two types of exceptions**
  - Exception due to an error, which stops the program (e.g., memory violation)
    - With speculation, result of program is undefined
    - Don’t want a speculative instruction to cause this
  - Exception that can be handled, allowing program to resume (e.g., page fault)
    - Result of program is defined
    - O.K. if a speculative instruction causes this
Speculative Execution

- Three methods for supporting speculation without introducing incorrect exception behavior
  - Hardware and OS ignore exceptions for speculative instructions
    - Programs that would terminate, may give incorrect results
  - A set of status bits called poison bits are attached to result registers written by speculated instructions when the instructions cause exceptions
    - The poison bits cause a fault when a normal instruction uses the register
    - More about this later (hardware support for compiler speculation)
  - Hardware support for speculation: buffer results from instructions until known that the instruction would execute
    - Reorder Buffer in extended Tomasulo’s algorithm

Hardware-based Speculation

- Modify dynamic scheduling logic to also support speculative execution
- In Tomasulo’s algorithm, 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 (write results) in order

- Key mechanism: Reorder Buffer
  - Hardware buffer for uncommitted results
Hardware-based Speculation: Reorder Buffer

- Reorder buffer can be operand source, if value not yet committed
- Once operand commits, result is found in register file

- Mechanism
  - At issue time, allocate an entry in the ROB to hold result
    - Each entry has 4 fields: instruction type, destination register, value, ready flag
  - Use ROB entry number instead of reservation station to rename
    - Since each value has a location in the ROB anyways
    - Can use additional registers for renaming, and ROB only for tracking commits
  - Instruction results commit to register set in order
    - Simple if ROB implemented as a queue
  - Undoing speculated instructions on mispredicted branches or on exceptions just requires throwing away uncommitted entries
    - Exceptions are not recognized until an instruction becomes ready to commit

 Four Steps of Speculative Tomasulo’s Algorithm

- **Issue**: Get instruction from the head of the instruction queue
  - If reservation station and ROB slot free, allocate and issue instruction
    - If operands are available, send them to the reservation station
    - If not, keep track of ROB entry that will produce the operands
  - If not free, stall issue

- **Execution**: Operate on operands (EX)
  - If both operands ready, execute
  - If not ready, watch CDB for result

- **Write result**: Finish execution (WB)
  - Write on CDB, mark reservation station available
  - Result picked up by ROB entry

- **Commit**: Update register with ROB result
  - When instruction at head of ROB and its result is present, update register (or send store to memory), free up ROB slot
  - If ROB head is an incorrectly predicted branch/faulting speculative instruction, flush ROB
Example of Tomasulo's Algorithm: Cycle 16

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Execute</th>
<th>Write</th>
<th>Name</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F6, 34(R2)</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>Load1</td>
<td>No</td>
</tr>
<tr>
<td>L.D F2, 45(R3)</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>Load2</td>
<td>No</td>
</tr>
<tr>
<td>MUL.D F0, F2, F4</td>
<td>3</td>
<td>15</td>
<td>16</td>
<td>10</td>
<td>Load3</td>
<td>No</td>
</tr>
<tr>
<td>SUB.D F8, F6, F2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>2</td>
<td>Mult1</td>
<td>No</td>
</tr>
<tr>
<td>DIV.D F10, F0, F6</td>
<td>5</td>
<td>40</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D F6, F8, F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load/Store Units

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vk</th>
<th>Qj</th>
<th>Qk</th>
<th>Rj</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add3</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>Mult2</td>
<td>Yes</td>
<td>DIV</td>
<td>#3</td>
<td>M(R4)</td>
<td>M(…)</td>
<td>Mult1</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Free Registers

<table>
<thead>
<tr>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
<th>F12</th>
<th>F16</th>
</tr>
</thead>
<tbody>
<tr>
<td>M(R4)</td>
<td>M(R3+45)</td>
<td></td>
<td>(M-M)+M</td>
<td>M(…)-M</td>
<td>Mult2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reorder Buffer

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Execute</th>
<th>Write</th>
<th>Commit</th>
<th>?</th>
<th>Instr.</th>
<th>Reg</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F6, 34(R2)</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>N</td>
<td>L</td>
<td>F6</td>
</tr>
<tr>
<td>L.D F2, 45(R3)</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>N</td>
<td>L</td>
<td>F2</td>
</tr>
<tr>
<td>MUL.D F0, F2, F4</td>
<td>3</td>
<td>15</td>
<td>16</td>
<td>10</td>
<td>3</td>
<td>Y</td>
<td>MUL</td>
<td>F0</td>
</tr>
<tr>
<td>SUB.D F8, F6, F2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>Y</td>
<td>SUB</td>
<td>F8</td>
</tr>
<tr>
<td>DIV.D F10, F0, F6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>Y</td>
<td>DIV</td>
<td>F10</td>
</tr>
<tr>
<td>ADD.D F6, F8, F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td>2</td>
<td>6</td>
<td>Y</td>
<td>ADD</td>
<td>F6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vk</th>
<th>Qj</th>
<th>Qk</th>
<th>Rj</th>
<th>Rk</th>
<th>Dest.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add3</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>Mult2</td>
<td>Yes</td>
<td>DIV</td>
<td>#3</td>
<td>M(…)</td>
<td>Mult1</td>
<td>#3</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Free Registers

<table>
<thead>
<tr>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
<th>F12</th>
<th>F16</th>
</tr>
</thead>
<tbody>
<tr>
<td>M(R3+45)</td>
<td>M(R2+34)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hardware-based Speculation

- Hardware-based speculation offers many advantages
  - Allows memory references to be disambiguated
  - Can incorporate hardware-based branch prediction
  - Maintains a precise exception model even for speculative instructions, since results don’t commit early
  - Does not require additional bookkeeping code
  - Does not depend on a good compiler

- Its main disadvantage is that it has substantial hardware requirements

- Schemes similar to what has been described have been implemented in the PowerPC 620, MIPS R10000, Intel P6, and AMD K5

Issuing Multiple Instructions Per Cycle

- Why?
  - All of the schemes described so far can at best achieve 1 instruction/cycle
  - Increasing transistor budgets, parallelism in instruction streams (independent instructions) have pushed for multiple instructions/cycle

Two variations

- **Superscalar**: varying number of instructions/cycle (1 to 8),
  - scheduled by a compiler (static) or by hardware (dynamic)
  - with or without speculation (implies hardware scheduling)
  - IBM Power2, Sun UltraSPARC, Pentium III/4, DEC Alpha, HP 8000

- **(Very) Long Instruction Words (V)L IW**: fixed set of instructions (4-16)
  - scheduled by the compiler: put ops into wide templates
  - i860, IA64 (with some hardware support)

- New metric of performance: Instructions Per Clock cycle (IPC) vs. CPI
Statically Scheduled Superscalar MIPS Processor

Superscalar MIPS: 2 instructions; 1 FP op, 1 other

- **Instruction issue**
  - Fetch 64-bits/clock cycle
    - Need to handle cache-line complications
  - Hardware determines whether 0, 1, or 2 instructions can be issued
    - Can only issue 2nd instruction if 1st instruction issues

- **Hazard detection**
  - Likelihood of hazards between two instructions in a packet
    - Simple solution: treat this as a structural hazard (issue only 1 of them)
  - 1 cycle load delay expands to 3 cycles in 2-way SS
    - Instruction in right half can’t use it, nor instructions in next slot
  - Branches also have a delay of 3 cycles

- **Execution**
  - Additional (or pipelined) functional units to derive benefit
  - 2 more ports for FP registers to do FP load or FP store and FP op

### 2-way Static Superscalar MIPS Pipeline

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Pipe Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Integer</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Integer</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Integer</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Pipeline Scheduling and Loop Unrolling

- Rare to find the ideal instruction mix of the previous slide
- Modern-day compilers apply several optimizations so as to expose Instruction Level Parallelism (ILP)

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 L.D F0, 0(R1)</td>
<td>1</td>
</tr>
<tr>
<td>stall</td>
<td>2</td>
</tr>
<tr>
<td>ADD.D F4, F0, F2</td>
<td>3</td>
</tr>
<tr>
<td>stall</td>
<td>4</td>
</tr>
<tr>
<td>S.D F4, 0(R1)</td>
<td>5</td>
</tr>
<tr>
<td>stall</td>
<td>6</td>
</tr>
<tr>
<td>DADDUI R1, R1, #−8</td>
<td>7</td>
</tr>
<tr>
<td>stall</td>
<td>8</td>
</tr>
<tr>
<td>BNE R1, R2, L1</td>
<td>9</td>
</tr>
<tr>
<td>stall</td>
<td>10</td>
</tr>
</tbody>
</table>

Pipeline Scheduling and Loop Unrolling (cont’d)

- **Loop unrolling** optimization: Replicate loop body multiple times, adjusting the loop termination code

```
L1: L.D F0, 0(R1)
    ADD.D F4, F0, F2
    S.D F4, 0(R1)
    ADD.D F6, −8(R1)
    S.D F8, F6, F2
    ADD.D F8, F6, F2
    S.D F8, −8(R1)
    L.D F10, −16(R1)
    ADD.D F12, F10, F2
    S.D F12, −16(R1)
    L.D F12, −24(R1)
    ADD.D F14, −24(R1)
    S.D F14, −8(R1)
    ADD.D F16, F14, F2
    S.D F16, F14, F2
    ADD.D F16, F14, F2
    L.D F16, 0(R1)
    S.D F4, −8(R1)
    DADDUI R1, R1, #−32
    S.D F12, 16(R1)
    BNE R1, R2, L1
    S.D F16, 8(R1)
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 L.D F0, 0(R1)</td>
<td>1</td>
</tr>
<tr>
<td>L.D F6, −8(R1)</td>
<td>2</td>
</tr>
<tr>
<td>L.D F10, −16(R1)</td>
<td>3</td>
</tr>
<tr>
<td>L.D F14, −24(R1)</td>
<td>4</td>
</tr>
<tr>
<td>ADD.D F4, F0, F2</td>
<td>5</td>
</tr>
<tr>
<td>ADD.D F8, F6, F2</td>
<td>6</td>
</tr>
<tr>
<td>ADD.D F12, F10, F2</td>
<td>7</td>
</tr>
<tr>
<td>ADD.D F16, F14, F2</td>
<td>8</td>
</tr>
<tr>
<td>S.D F4, 0(R1)</td>
<td>9</td>
</tr>
<tr>
<td>S.D F4, −8(R1)</td>
<td>10</td>
</tr>
<tr>
<td>DADDUI R1, R1, #−32</td>
<td>11</td>
</tr>
<tr>
<td>S.D F12, 16(R1)</td>
<td>12</td>
</tr>
<tr>
<td>BNE R1, R2, L1</td>
<td>13</td>
</tr>
<tr>
<td>S.D F16, 8(R1)</td>
<td>14</td>
</tr>
</tbody>
</table>
Loop Unrolling for Superscalar Processors

- Unroll loop 5 times to avoid extra 1-cycle delays in 2-way SS

<table>
<thead>
<tr>
<th>L1:</th>
<th>Integer Instruction</th>
<th>FP Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>L.D F0, 0(R1)</td>
<td>1</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
<td>L.D F6, -8(R1)</td>
</tr>
<tr>
<td>S.D</td>
<td>F4, 0(R1)</td>
<td>L.D F10, -16(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F6, -8(R1)</td>
<td>ADD.D F4, F0, F2</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F8, F6, F2</td>
<td>L.D F14, -24(R1)</td>
</tr>
<tr>
<td>S.D</td>
<td>F8, -8(R1)</td>
<td>ADD.D F8, F6, F2</td>
</tr>
<tr>
<td>L.D</td>
<td>F10, -16(R1)</td>
<td>L.D F18, -32(R1)</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F12, F10, F2</td>
<td>ADD.D F12, F10, F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F12, -16(R1)</td>
<td>S.D F4, -8(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F14, -24(R1)</td>
<td>ADD.D F20, F18, F2</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F16, F14, F2</td>
<td>S.D F12, -16(R1)</td>
</tr>
<tr>
<td>S.D</td>
<td>F16, -24(R1)</td>
<td>S.D F16, 16(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F18, -32(R1)</td>
<td>BNE R1, R2, L1</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F20, F18, F2</td>
<td>S.D F20, 8(R1)</td>
</tr>
<tr>
<td>S.D</td>
<td>F20, -32(R1)</td>
<td></td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1, #40</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>R1, R2, #40</td>
<td></td>
</tr>
</tbody>
</table>

Loop unrolling: 10 (actually 6) to 3.5 cycles/iteration
SS: 3.5 cycles/iteration to 2.4 (1.5 times improvement)

Dynamic Scheduling in Superscalar Processors

- How to extend Tomasulo’s algorithm?
  - Simple solution
    - Assume 1 integer + 1 floating point FU
    - Separate Tomasulo control for integer and for floating point
    - Need to handle the fact that FP loads can cause dependency between integer and FP issue
      - Replace load reservation station with a load queue;
      - Load checks addresses in Store Queue to avoid RAW violation
      - Store checks addresses in Load & Store Queues to avoid WAR,WAW
      - Called “decoupled architecture”
  - General solution
    - Allow issue stage to work faster than rest of architecture
      - Achieved by both pipelining and widening the issue logic
      - Instructions issued, reservation stations allocated in order
    - Rest of the design already supports overlapped execution
      - Need wider CDB to store multiple results/cycle
Limits of Superscalar Processors

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards

- Need more instructions to issue at the same time to get improved performance
  - However, greater difficulty of decode and issue
  - Even 2-way superscalar => examine 2 opcodes, 6 register specifiers, and decide if 1 or 2 instructions can issue

- Issue rates of modern processors vary between 2—8 instructions/cycle

- Motivation for VLIW and EPIC processors
  - Hardware is simple: issues the packet given it by the compiler
  - Next lecture

Limits to Multi-Issue Machines

- Limitations specific to either SS or VLIW implementation
  - Decode/issue in SS
  - VLIW code size: unroll loops + wasted fields in VLIW
  - VLIW lock step => 1 hazard and all instructions stall
  - VLIW and binary compatibility is practical weakness

- Inherent limitations of ILP
  - 1 branch in 5 instructions => how to keep a 5-way VLIW busy?
  - Latencies of units => many operations must be scheduled
  - Need on the order of Pipeline Depth x No. Functional Units of independent instructions to keep all busy

- Is there that much parallelism in programs?
Studies of the Limitations of ILP

- Start off with a hardware model of an ideal processor
  1. Register renaming –infinite virtual registers and all WAW & WAR hazards are avoided
  2. Branch prediction –perfect; no mispredictions
  3. Jump prediction –all jumps perfectly predicted => machine with perfect speculation and an unbounded buffer of instructions available (predicts address)
  4. Memory-address alias analysis –addresses are known & a store can be moved before a load provided addresses not equal

- 1 cycle latency for all instructions

Upper Limit to ILP for Six SPEC92 Benchmarks

![Graph showing instruction issues per cycle for different SPEC benchmarks.]