Amdahl’s Law

\[
\text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times \left[1 - \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right] + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}
\]

\[
\text{Speedup}_{\text{overall}} = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{(1 - \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}
\]
Review: Performance

<table>
<thead>
<tr>
<th>CPU time</th>
<th>Seconds</th>
<th>Instructions</th>
<th>Cycles</th>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Program</td>
<td>Instruction</td>
<td>Cycle</td>
<td></td>
</tr>
</tbody>
</table>

“Average Cycles Per Instruction”

\[
\text{CPI} = \frac{\text{CPU time} \times \text{Clock Rate}}{\text{Instruction Count}} = \frac{\text{Cycles}}{\text{Instruction Count}}
\]

\[
\text{CPU time} = \text{Cycle Time} \times \sum_{i=1}^{n} \text{CPI}_i \times I_i
\]

“Instruction Frequency”

\[
\text{CPI} = \sum_{i=1}^{n} \text{CPI}_i \times F_i \quad \text{where } F_i = \frac{I_i}{\text{Instruction Count}}
\]

Invest Resources where time is Spent!

Example Solution

\[
\text{Exec Time} = \text{Instr Cnt} \times \text{CPI} \times \text{Clock}
\]

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>Freq</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>.50</td>
<td>.5</td>
<td>.5 - X</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>.20</td>
<td>.4</td>
<td>.2 - X</td>
<td>2</td>
</tr>
<tr>
<td>Store</td>
<td>.10</td>
<td>.2</td>
<td>.1</td>
<td>2</td>
</tr>
<tr>
<td>Branch</td>
<td>.20</td>
<td>.3</td>
<td>.2</td>
<td>3</td>
</tr>
<tr>
<td>Reg/Mem</td>
<td>X</td>
<td></td>
<td>1 - X</td>
<td></td>
</tr>
</tbody>
</table>

\[
1.00 \times 1.5 = (1 - X) \times (1.7 - X)/(1 - X)
\]

Instr Cnt_{Old} \times CPI_{Old} \times Clock_{Old} = Instr Cnt_{New} \times CPI_{New} \times Clock_{New}

\[
1.00 \times 1.5 = (1 - X) \times (1.7 - X)/(1 - X)
\]

All loads must be eliminated for this to be a win!
What Does the Mean Mean?

- **Arithmetic mean (AM):** (weighted arithmetic mean) tracks execution time: \( \sum_{1..N} (\text{Time}_i)/N \) or \( \sum (W_i*\text{Time}_i) \)

- **Harmonic mean (HM):** (weighted harmonic mean) of rates (e.g., MFLOPS) tracks execution time: \( N/ \sum_{1..N} (1/\text{Rate}_i) \) or \( 1/ \sum (W_i/\text{Rate}_i) \)
  - Arithmetic mean cannot be used for rates (e.g., IPC)
  - 30 MPH for 1 mile + 90 MPH for 1 mile != avg 60 MPH

- **Geometric mean (GM):** average speedups of \( N \) programs
  \( N\sqrt[N]{} \prod_{1..N} (\text{speedup}(i)) \)

---

Little’s Law

- **Key Relationship between latency and bandwidth:**
- **Average number in system = arrival rate * mean holding time**

- **Example:**
  - How big a wine cellar should we build?
  - We drink (and buy) an average of 4 bottles per week
  - On average, I want to age my wine 5 years
  - bottles in cellar = 4 bottles/week \* 52 weeks/year \* 5 years
  - = 1040 bottles
Instruction Sets

• Basic classes
  – Stack, accumulator, General purpose register

• Impact on implementation

• Impact on compiler

• Transmeta approach

Review: The Five Stages of a Load

• Ifetch: Instruction Fetch
  – Fetch the instruction from the Instruction Memory

• Reg/Dec: Registers Fetch and Instruction Decode

• Exec: Calculate the memory address

• Mem: Read the data from the Data Memory

• WrB: Write the data back to the register file
Its Not That Easy for Computers

• What could go wrong?
• Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  – **Structural hazards**: HW cannot support this combination of instructions
  – **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
    » RAW
    » WAW
    » WAR
  – **Control hazards**: Pipelining of branches & other instructions

---

**Control Hazard**

• **Although Beq is fetched during Cycle 4:**
  – Target address is **NOT** written into the PC until the **end of Cycle 7**
  – Branch’s target is **NOT** fetched until **Cycle 8**
  – 3-instruction delay before the branch take effect
• **This is called a Control Hazard:**
Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken
   – Execute successor instructions in sequence
   – “Squash” instructions in pipeline if branch actually taken
   – Advantage of late pipeline state update
   – PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken
   – Need to compute branch target

#4: Delayed Branch
   – Define branch to take place AFTER a following instruction

Dynamic Branch Prediction

- Solution: 2-bit counter where prediction changes only if mispredict twice:
- Increment for taken, decrement for not-taken
  - 00, 01, 10, 11
- Helps when target is known before condition
Correlating Branches

Idea: taken/not taken of recently executed branches is related to behavior of next branch (as well as the history of that branch behavior)

- Then behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction

Branch address

2-bits per branch predictor

2-bit global branch history

Need Address @ Same Time as Prediction

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address (Figure 4.22, p. 273)

Procedure Return Addresses Predicted with a Stack
Hybrid/Competitive/Selective Branch Predictor

- Different predictors work better for different branches
- Pick the predictor that works best for a given branch

FP Loop Showing Stalls

1. Loop: LD F0,0(R1) ; F0=vector element
2. stall
3. ADDD F4,F0,F2 ; add scalar in F2
4. stall
5. stall
6. SD 0(R1),F4 ; store result
7. SUBI R1,R1,8 ; decrement pointer 8B (DW)
8. BNEZ R1,Loop ; branch R1!=zero
9. stall ; delayed branch slot

- Rewrite code to minimize stalls?

<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>
</tbody>
</table>
Revised FP Loop Minimizing Stalls

1. Loop:  
   1. LD  F0,0(R1)
   2. stall
   3. ADDD F4,F0,F2
   4. SUBI R1,R1,8
   5. BNEZ R1,Loop ;delayed branch
   6. SD  8(R1),F4 ;altered when move past SUBI

<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>
</tbody>
</table>

How do we make this faster?

Unroll Loop Four Times

1. Loop:  
   1. LD  F0,0(R1)
   2. ADDD F4,F0,F2
   3. SD  0(R1),F4 ;drop SUBI & BNEZ
   4. LD  F6,-8(R1)
   5. ADDD F8,F6,F2
   6. SD  -8(R1),F8 ;drop SUBI & BNEZ
   7. LD  F10,-16(R1)
   8. ADDD F12,F10,F2
   9. SD  -16(R1),F12 ;drop SUBI & BNEZ
  10. LD  F14,-24(R1)
  11. ADDD F16,F14,F2
  12. SD  -24(R1),F16
  13. SUBI R1,R1,#32 ;alter to 4*8
  14. BNEZ R1,LOOP
  15. NOP

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration

Assumes R1 is multiple of 4
Unrolled Loop That Minimizes Stalls

• What assumptions made when moved code?
  – OK to move store past SUBI even though changes register
  – OK to move loads before stores: get right data?
  – When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration

SW Pipelining Example

Before: Unrolled 3 times

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

After: Software Pipelined

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD</td>
<td>F0,0(R1)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADDDD</td>
<td>F4,F0,F2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LD</td>
<td>F0,-8(R1)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ADDDD</td>
<td>F4,F0,F2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SD</td>
<td>0(R1),F4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>SD</td>
<td>-8(R1),F4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LD</td>
<td>F10,-16(R1)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ADDDD</td>
<td>F12,F10,F2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>SUBI</td>
<td>R1,R1,#32</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>BNEZ</td>
<td>R1,LOOP</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>SD</td>
<td>0(R1),F4</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>AD</td>
<td>ADD</td>
<td>F4,F0,F2</td>
</tr>
<tr>
<td>13</td>
<td>LD</td>
<td>F0,-16(R1)</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>LD</td>
<td>F16,F14,F2</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>BNEZ</td>
<td>R1,LOOP</td>
<td></td>
</tr>
</tbody>
</table>

14 clock cycles, or 3.5 per iteration
SW Pipelining Example

Symbolic Loop Unrolling

- *Less code space*
- Overhead paid only once vs. each iteration in loop unrolling

<table>
<thead>
<tr>
<th>Software Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Overlapped Operations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop Unrolling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Overlapped Operations</td>
</tr>
</tbody>
</table>

Overlap between unrolled iters

Proportional to number of unrolls

100 iterations = 25 loops with 4 unrolled iterations each

Four Stages of Scoreboard Control

1. **Issue**: decode instructions & check for structural hazards (ID1)
   
   If a functional unit for the instruction is free and no other active instruction has the same destination register (WAW), the scoreboard issues the instruction to the functional unit and updates its internal data structure. If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.

2. **Read operands**: wait until no data hazards, then read operands (ID2)
   
   A source operand is available if no earlier issued active instruction is going to write it, or if the register containing the operand is being written by a currently active functional unit. When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order.
Four Stages of Scoreboard Control

3. Execution: operate on operands
   The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.

4. Write Result: finish execution (WB)
   Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards. If none, it writes results. If WAR, then it stalls the instruction.
   Example:
   
<table>
<thead>
<tr>
<th>Instruction</th>
<th>FP1</th>
<th>FP2</th>
<th>FP3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVD</td>
<td>F0</td>
<td>F2</td>
<td>F4</td>
</tr>
<tr>
<td>ADDD</td>
<td>F10</td>
<td>F0</td>
<td>F8</td>
</tr>
<tr>
<td>SUBD</td>
<td>F8</td>
<td>F8</td>
<td>F14</td>
</tr>
</tbody>
</table>

   CDC 6600 scoreboard would stall SUBD until ADDD reads operands

Tomasulo Organization

[Diagram of Tomasulo Organization]

Common Data Bus (CDB)
Tomasulo Summary

• Prevents Register as bottleneck
• Avoids WAR, WAW hazards of Scoreboard
• Allows loop unrolling in HW
• Not limited to basic blocks (provided branch prediction)

• Lasting Contributions
  – Dynamic scheduling
  – Register renaming
  – Load/store disambiguation

Speculation (getting 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” squash)
• Often combine with dynamic scheduling
• 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**
• Memory operations (memory disambiguation)
• Interrupts -> maintaining precise exceptions
HW support for More ILP

- 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, it's easy to undo speculated instructions on mispredicted branches or on exceptions

Recovering from Incorrect Speculation

- Reorder Buffer
  - Register Update Unit: Reorder buffer+reservation stations combined
  - P6 Style: Reorder buffer separate from reservation stations
- R10K style
  - Separate physical register file from reorder buffer
  - Must maintain a map of logical to physical registers
- Enables easy recovery from misprediction & exceptions
- Memory Disambiguation
  - Load/store queue (Memory Order Buffer)
Superscalar & VLIW

- Wider pipelines
- Superscalar, multiple PCs
- VLIW, multiple operations for each PC
- Problems w/ Superscalar
  - Wide fetch
  - Dependence check
  - Bypassing
  - Need large window to find independent ops

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
Trace Scheduling

Reorder these instructions to improve ILP

Fix-up instructions
In case we were wrong

Trace Cache

- Store traces
- Enables fetch past next branch
- Enables branch folding
Predicated/Conditional Execution (more ILP)

- Avoid branch prediction by turning branches into conditionally executed instructions:
  
  \[
  \text{if (x) then } A = B \text{ op } C \text{ else NOP}
  \]
  
  - If false, then neither store result nor cause exception
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr, IA-64 predicated execution.

- Drawbacks to conditional instructions
  - Still takes a clock even if “annulled”
  - Stall if condition evaluated late
  - Complex conditions reduce effectiveness; condition becomes known late in pipeline

Other Topics

- Power as Design target
- Reliability (Diva approach)