Lecture 14

- Halt/term structure
- Today's lecture:
  - Finite stalls
  - Branches
  - Another look at performance
- Why do we need to stall sometimes?

Stalling

- The easiest solution is to stall the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble.

Stalling delays the entire pipeline

- If we delay the second instruction, we'll have to delay the third one too.
- It prevents problems such as two instructions trying to write to the same register in the same cycle.
- Also allows forwarding between AND and OR.

Detected stalls, cont.

- When should stalls be detected?
  - `lw $1, 20($3)`
  - `and $12, $1, $5`
  - `or $13, $12, $2`

- What is the stall condition?
  - `if ( $5 = 0x00000000 ) {` (in C)
    - `then stall`
Branches in the original pipelined datapath

Stalling is one solution

- Again, stalling is always one possible solution.

\[
\begin{array}{cccccccc}
\text{Clock cycle} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
\text{beq $s2, s3, Label$} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} \\
\end{array}
\]

- Here we stall until cycle 4, after we do make the branch decision.

Branch misprediction

- If our guess is wrong, then we would have already started executing two instructions incorrectly. We’ll have to discard, or flush, these instructions and begin executing the right ones from the branch target address, Label.

\[
\begin{array}{cccccccc}
\text{Clock cycle} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
\text{beq $s2, s3, Label$} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} \\
\end{array}
\]

Branch prediction

- Another approach is to guess whether or not the branch is taken.
- In terms of hardware, this is done via a branch predictor.
- This way we just pretend the PC will continue executing, as for normal instructions.
- If we’re correct, then there is no problem and the pipeline keeps going at full speed.

\[
\begin{array}{cccccccc}
\text{Clock cycle} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
\text{beq $s2, s3, Label$} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} & \text{??} \\
\end{array}
\]

Performance gains and losses

- Overall, branch prediction is worth it.
- Mispredicting a branch means that two clock cycles are wasted.
- But if our predictions are even just occasionally correct, then this is preferable to stalling and wasting two cycles for every branch.
- Most modern CPUs use branch prediction.
- Accurate predictions are important for optimal performance.
- More CPUs predict branches dynamically—statistics are kept at runtime to determine the likelihood of a branch being taken.
- The pipeline structure also has a big impact on branch prediction.
- A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.
- We must also be careful that instructions do not modify registers or memory before they get flushed.
Implementing branches

- We can actually decide the branch a little earlier, in ID instead of EX.
  - Our sample instruction set has only a 802.
- Then we would only need to flush one instruction on a misprediction.

Branching without forwarding and load stalls

Summary of Hazards/Stalls/Branches

- Three kinds of hazards conspire to make pipelining difficult.
  - Structural hazards result from not having enough hardware available to execute multiple instructions simultaneously.
  - Data hazards occur when instructions need to access registers that haven’t been updated yet.
  - Control hazards arise when the CPU cannot determine which instruction to fetch next.

Performance

- Now we’ll discuss issues related to performance:
  - Latency/Response Time/Execution Time vs. Throughput
  - How do you make a reasonable performance comparison?
  - The 3 components of CPU performance
  - The 2 laws of performance
Why know about performance

- **Purchasing Perspective:**
  - Given a collection of machines, which has the
    - Best Performance?
    - Lowest Price?
    - Best Performance/Price?

- **Design Perspective:**
  - Faced with design options, which has the
    - Best Performance Improvement?
    - Lowest Cost?
    - Best Performance/Cost?

- **Both require**
  - Basis for comparison
  - Metric for evaluation

---

Two notions of performance

<table>
<thead>
<tr>
<th>Plane</th>
<th>DC to Paris</th>
<th>Speed</th>
<th>Passengers</th>
<th>Throughput (gpm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>747</td>
<td>6.5 hours</td>
<td>610 m/h</td>
<td>470</td>
<td>284,700</td>
</tr>
<tr>
<td>Concorde</td>
<td>3 hours</td>
<td>1350 m/h</td>
<td>132</td>
<td>178,200</td>
</tr>
</tbody>
</table>

- Which has higher performance?
  - Depends on the metric
    - Time to do the task (Execution Time, Latency, Response Time)
    - Throughput Bandwidth
    - Response time and throughput are often in opposition

---

Some Definitions

- **Performance is in units of thing/unit time**
  - E.g., Nemburs/hr
    - Bigger is better

- If we are primarily concerned with response time
  - \[ \text{Performance}(x) = \frac{1}{\text{execution_time}(x)} \]

- Relative performance: \( x \times t \) times faster than \( y \)
  - \[ N = \text{Performance}(x) \times \text{execution_time}(y) \]
    \[ \text{Performance}(x) \times \text{execution_time}(y) \]

---

Basis of Comparison

- When comparing systems, need to fix the workload
  - Which workload?

<table>
<thead>
<tr>
<th>Workload</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Target</td>
<td>Representative</td>
<td>Very specific</td>
</tr>
<tr>
<td>Workload</td>
<td></td>
<td>Not representative</td>
</tr>
<tr>
<td>Full application benchmarks</td>
<td>Portable</td>
<td>Less representative</td>
</tr>
<tr>
<td>Kernel or benchmarks</td>
<td>Easy to run</td>
<td>Useful early in design</td>
</tr>
<tr>
<td>Microbenchmarks</td>
<td>Identify peak capability and potential bottlenecks</td>
<td>Real application performance may be much below peak</td>
</tr>
</tbody>
</table>

---

Benchmarking

- Some common benchmarks include:
  - Adobe Photoshop for image processing
  - BAPCo Sysmark for office applications
  - Unreal Tournament 2003 for 3D games
  - SPEC2000 for CPU performance

- The best way to see how a system performs for a variety of programs is to just show the execution times of all the programs.
- Here are execution times for several different Photoshop 5.5 tasks; from http://www.tech-report.com
Summarizing performance

- Summarizing performance with a single number can be misleading—just like summarizing four years of school with a single GPA!
- If you must have a single number, you could sum the execution times.
  This example graph displays the total execution time of the individual tests from the previous page.
- A similar option is to find the average of all the execution times.
  For example, the 800MHz Pentium III (in yellow) needed 227.3 seconds to run 21 programs, so its average execution time is 227.3 / 21 = 10.82 seconds.
- A weighted sum or average is also possible, and lets you emphasize some benchmarks more than others.

Three Components of CPU Performance

\[ \text{CPU time}_{\text{cy}} = \text{instructions executed} \times \text{CPI} \times \text{clock cycle time} \]

Cycles Per Instruction

CPI (Review)

- The average number of clock cycles per instruction, or CPI, is a function of the machine and program.
  - The CPI depends on the actual instructions appearing in the program—a floating-point intensive application might have a higher CPI than an integer-based program.
  - It also depends on the CPU implementation. For example, a Pentium can execute the same instructions as an older 80486, but faster.
- Initially we assumed each instruction took one cycle, so we had CPI = 1.
  - The CPI can be > 1 due to memory stalls and slow instructions.
  - The CPI can be < 1 on machines that execute more than 1 instruction per cycle (superscalar).

The components of execution time

- Execution time can be divided into two parts.
  - User time is spent running the application program itself.
  - System time is when the application calls operating system code.
- The distinction between user and system time is not always clear, especially under different operating systems.
- The Unix time command shows both.

Instructions Executed

- Instructions executed:
  - We are not interested in the static instruction count, or how many lines of code are in a program.
  - Instead we care about the dynamic instruction count, or how many instructions are actually executed when the program runs.
- There are three lines of code below, but the number of instructions executed would be 2001:

```
T1  Sub $a0, 1000
Ostrich: sub $a0, $a0, 1
      bne $a0, 0, ostrich
```

Execution time, again

\[ \text{CPU time}_{\text{cy}} = \text{instructions executed} \times \text{CPI} \times \text{clock cycle time} \]

- The easiest way to remember this is match up the units:

<table>
<thead>
<tr>
<th>Program</th>
<th>Compiler</th>
<th>ISA</th>
<th>Organization</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executed</td>
<td>CPI</td>
<td>Clock Cycle Time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Make things faster by making any component smaller!!

- Often easy to reduce one component by increasing another.
Example: Comparing across ISAs

- Intel's Itanium (64-bit) ISA is designed to facilitate executing multiple instructions per cycle. If an Itanium processor achieves an average CPI of 3.3 instructions per cycle, how much faster is it than a Pentium (which uses the 386 ISA) with an average CPI of 17 (assume same freq.):
  a) Itanium is three times faster
  b) Itanium is one third as fast
  c) Not enough information

Example: CPI Improvements

- Base Machine:

<table>
<thead>
<tr>
<th>Op Type</th>
<th>Freq (%)</th>
<th>Cycles</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

- How much faster would the machine be if:
  - we added a cache to reduce average load time to 3 cycles?
  - we added a branch predictor to reduce branch time by 1 cycle?
  - we could do two ALU operations in parallel?

Improving CPI

- Many processor design techniques we'll see improve CPI
  - Often they only improve CPI for certain types of instructions
  - \( \text{CPI} = \sum_{i=1}^{n} \text{CPI}_i = F_i \) where \( F_i = \frac{l_i}{C_i} \) Instruction Count
  - \( F_i \) = Fraction of instructions of type \( i \)

- First Law of Performance:
  Make the common case fast

Amdahl's Law

- Amdahl's Law states that optimizations are limited in their effectiveness.

  \[ \frac{\text{Time affected by improvement}}{\text{Amount of improvement}} + \frac{\text{Time unaffected by improvement}}{\text{Amount of improvement}} = 1 \]

- For example, doubling the speed of floating-point operations sounds like a great idea. But if only 10% of the program execution time \( T \) involves floating point code, then the overall performance improves by just 5%.

  \[ \text{Execution time after improvement} = \frac{0.10 T}{2} + 0.90 T = 0.95 T \]

- What is the maximum speedup from improving floating point?

- Second Law of Performance:
  Make the fast case common

Summary

- Performance is one of the most important criteria in judging systems.
- There are two main measurements of performance.
  - Execution time is what we'll focus on.
  - Throughput is important for servers and operating systems.
- Our main performance equation explains how performance depends on several factors related to both hardware and software.

  \[ \text{CPU time} = \text{Instructions executed} \times \text{CPU time per instruction} \]

- It can be hard to measure these factors in real life, but this is a useful guide for comparing systems and designs.
- Amdahl's Law tells us how much improvement we can expect from specific enhancements.
- The best benchmarks are real programs, which are more likely to reflect common instruction mixes.