Review: Moore’s Law

- “Cramming More Components onto Integrated Circuits”
  - Gordon Moore, Electronics, 1965
- # on transistors on cost-effective integrated circuit double every 18 months

Review: Joy’s Law in ManyCore world

- VAX: 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

Review: Categories of Thread Execution

- Superscalar
- Fine-Grained
- Coarse-Grained
- Multiprocessing
- Multithreading

“Bell’s Law” – new class per decade

- Enabled by technological opportunities
- Smaller, more numerous and more intimately connected
- Brings in a new kind of application
- Used in many ways not previously imagined

Metrics used to Compare Designs

- Cost
  - Die cost and system cost
- Execution Time
  - average and worst-case
  - Latency vs. Throughput
- Energy and Power
  - Also peak power and peak switching current
- Reliability
  - Resiliency to electrical noise, part failure
  - Robustness to bad software, operator error
- Maintainability
  - System administration costs
- Compatibility
  - Software costs dominate

What is Performance?

- Latency (or response time or execution time)
  - time to complete one task
- Bandwidth (or throughput)
  - tasks completed per unit time

Today:
Quick review of everything you should have learned

(A countably-infinite set of computer architecture concepts)
**Definition: Performance**

- Performance is in units of things per sec
  - bigger is better
- If we are primarily concerned with response time

\[
\text{performance}(x) = \frac{1}{\text{execution_time}(x)}
\]

"X is n times faster than Y" means

\[
n = \frac{\text{Performance}(X)}{\text{Performance}(Y)} = \frac{\text{Execution_time}(Y)}{\text{Execution_time}(X)}
\]

---

**Performance: What to measure**

- Usually rely on benchmarks vs. real workloads
- To increase predictability, collections of benchmark applications—*benchmark suites*—are popular
- **SPECCPU**: popular desktop benchmark suite
  - CPU only, split between integer and floating point programs
  - SPECint2000 has 12 integer, SPECfp2000 has 14 integer pgms
  - SPECCPU2006 to be announced Spring 2006
  - SPECSFS (NFS file server) and SPECWeb (WebServer) added as server benchmarks
- **Transaction Processing Council** measures server performance and cost-performance for databases
  - TPC-C Complex query for Online Transaction Processing
  - TPC-H models ad hoc decision support
  - TPC-W a transactional web benchmark
  - TPC-App application server and web services benchmark

---

**Summarizing Performance**

Which system is faster?

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

---

... depends who’s selling

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

Average throughput

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.50</td>
<td>2.00</td>
<td>1.25</td>
</tr>
<tr>
<td>B</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Throughput relative to B

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>B</td>
<td>2.00</td>
<td>0.50</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Throughput relative to A
Summarizing Performance over Set of Benchmark Programs

Arithmetic mean of execution times $t_i$ (in seconds)
$$1/n \sum_i t_i$$

Harmonic mean of execution rates $r_i$ (MIPS/MFLOPS)
$$n/ \left[ \sum_i (1/r_i) \right]$$

• Both equivalent to workload where each program is run the same number of times
• Can add weighting factors to model other workload distributions

Normalized Execution Time and Geometric Mean

• Measure speedup up relative to reference machine
  $$\text{ratio} = t_{\text{Ref}}/t_A$$
• Average time ratios using geometric mean
  $$n\sqrt[n]{\prod_i \text{ratio}_i}$$
• Insensitive to machine chosen as reference
• Insensitive to run time of individual benchmarks
• Used by SPEC89, SPEC92, SPEC95, …, SPEC2006

……. But beware that choice of reference machine can suggest what is “normal” performance profile:

Vector/Superscalar Speedup

• 100 MHz Cray J90 vector machine versus 300MHz Alpha 21164
• [LANL Computational Physics Codes, Wasserman, ICS’96]
• Vector machine peaks on a few codes????

Superscalar/Vector Speedup

• 100 MHz Cray J90 vector machine versus 300MHz Alpha 21164
• [LANL Computational Physics Codes, Wasserman, ICS’96]
• Scalar machine peaks on one code???
How to Mislead with Performance Reports

- Select pieces of workload that work well on your design, ignore others
- Use unrealistic data set sizes for application (too big or too small)
- Report throughput numbers for a latency benchmark
- Report latency numbers for a throughput benchmark
- Report performance on a kernel and claim it represents an entire application
- Use 16-bit fixed-point arithmetic (because it’s fastest on your system) even though application requires 64-bit floating-point arithmetic
- Use a less efficient algorithm on the competing machine
- Report speedup for an inefficient algorithm (bubblesort)
- Compare hand-optimized assembly code with unoptimized C code
- Compare your design using next year’s technology against competitor’s year old design (1% performance improvement per week)
- Ignore the relative cost of the systems being compared
- Report averages and not individual results
- Report speedup over unspecified base system, not absolute times
- Report efficiency not absolute times
- Report MFLOPS not absolute times (use inefficient algorithm)

[David Bailey “Twelve ways to fool the masses when giving performance results for parallel supercomputers”]

Amdahl’s Law

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

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

Best you could ever hope to do:

\[
\text{Speedup}_{\text{maximum}} = \frac{1}{1 - \text{Fraction}_{\text{enhanced}}}
\]

Amdahl’s Law example

- New CPU 10X faster
- I/O bound server, so 60% time waiting for I/O

\[
\text{Speedup}_{\text{overall}} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = \frac{1}{0.64} = 1.56
\]

- Apparently, its human nature to be attracted by 10X faster, vs. keeping in perspective its just 1.6X faster

Computer Performance

<table>
<thead>
<tr>
<th>Inst Count</th>
<th>CPI</th>
<th>Clock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>X (X)</td>
<td></td>
</tr>
<tr>
<td>Inst. Set.</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Organization</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Technology</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Cycles Per Instruction (Throughput)

"Average Cycles per Instruction"

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

\[
\text{CPU time} = \text{Cycle Time} \times \sum \text{CPI}_j \times I_j
\]

\[
CPI = \sum \text{CPI}_j \times F_j \quad \text{where } F_j = \frac{I_j}{\text{Instruction Count}}
\]

"Instruction Frequency"

Example: Calculating CPI bottom up

Run benchmark and collect workload characterization (simulate, machine counters, or sampling)

<table>
<thead>
<tr>
<th>Base Machine (Reg / Reg)</th>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>(% Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>(33%)</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>(27%)</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>2</td>
<td>.2</td>
<td>(13%)</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>(27%)</td>
<td></td>
</tr>
</tbody>
</table>

Typical Mix of instruction types in program

Design guideline: Make the common case fast

MIPS 1% rule: only consider adding an instruction of it is shown to add 1% performance improvement on reasonable benchmarks.

Power and Energy

- Energy to complete operation (Joules)
  - Corresponds approximately to battery life
  - (Battery energy capacity actually depends on rate of discharge)

- Peak power dissipation (Watts = Joules/second)
  - Affects packaging (power and ground pins, thermal design)

- \(\frac{\text{di/dt}}{\text{Amps/second}}\), peak change in supply current
  - Affects power supply noise (power and ground pins, decoupling capacitors)

Peak Power versus Lower Energy

- System A has higher peak power, but lower total energy
- System B has lower peak power, but higher total energy
CS 252 Administrivia

• Sign up!  Web site is:  http://www.cs.berkeley.edu/~kubitron/cs252
• Review: Chapter 1, Appendix A, B, C
• CS 152 home page, maybe “Computer Organization and Design (COD)2/e”
  – If did take a class, be sure COD Chapters 2, 5, 6, 7 are familiar
  – Copies in Bechtel Library on 2-hour reserve
• First two readings are up (look on Lecture page)
  – Read the assignment carefully, since the requirements vary about what you need to turn in
  – Submit results to website before class
    » (will be a link up on handouts page)
  – You can have 5 total late days on assignments
    » 10% per day afterwards
    » Save late days!

ISA Implementation Review

A "Typical" RISC ISA

• 32-bit fixed format instruction (3 formats)
• 32 32-bit GPR (R0 contains zero, DP take pair)
• 3-address, reg-reg arithmetic instruction
• Single address mode for load/store: base + displacement
  – no indirection
• Simple branch conditions
• Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC,
    CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Example: MIPS (- MIPS)

Register-Register

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td>Op</td>
</tr>
</tbody>
</table>
```

Register-Immediate

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Op</td>
<td>Rs1</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>
```

Branch

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Op</td>
<td>Rs1</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>
```

Jump / Call

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>target</td>
</tr>
</tbody>
</table>
```

Datapath vs Control

- **Datapath:** Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals
- **Controller:** State machine to orchestrate operation on the data path
  - Based on desired function and signals

5 Steps of MIPS Datapath

1. **Instruction Fetch**
   - Next PC

2. **Instr. Reg. Fetch**
   - IR <= mem[PC];
   - PC <= PC + 4
   - A <= Reg[IR_rs];
   - B <= Reg[IR_rt]

3. **Execute Addr. Calc**
   - rs1t <= A oP_{inst} B
   - WB <= rs1t
   - Reg[IR_ra] <= WB

4. **Memory Access**
   - Next SEQ PC

5. **Write Back**
   - Reg[IR_rd] <= WB

**Data stationary control**
- local decode for each instruction phase / pipeline stage
Visualizing Pipelining

Figure A.2, Page A-8

Pipelining is not quite that easy!

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

One Memory Port/Structural Hazards

Figure A.4, Page A-14

(Similar to Figure A.5, Page A-15)

How do you “bubble” the pipe?
**Speed Up Equation for Pipelining**

\[
CPI_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}
\]

\[
\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]

For simple RISC pipeline, \(CPI = 1\):

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]

**Example: Dual-port vs. Single-port**

- Machine A: Dual ported memory (“Harvard Architecture”)
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

\[
\begin{align*}
\text{SpeedUp}_A &= \frac{\text{Pipeline Depth}}{1} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}} \\
\text{SpeedUp}_B &= \frac{\text{Pipeline Depth}}{1 + 0.4} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{unpipe}} / 1.05} \\
&= \frac{\text{Pipeline Depth}}{1.4} \times 1.05 \\
&= 0.75 \times \text{Pipeline Depth}
\end{align*}
\]

\[
\frac{\text{SpeedUp}_A}{\text{SpeedUp}_B} = \frac{\text{Pipeline Depth}}{0.75 \times \text{Pipeline Depth}} = 1.33
\]

- Machine A is 1.33 times faster

**Data Hazard on R1**

- Read After Write (RAW)
  - Instr\(_J\) tries to read operand before Instr\(_I\) writes it
  - Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.

**Three Generic Data Hazards**

- Instr\(_I\): add \(r1, r2, r3\)
- Instr\(_J\): sub \(r4, r1, r3\)
Three Generic Data Hazards

• Write After Read (WAR)
  Instr\textsubscript{j} writes operand \textit{before} Instr\textsubscript{i} reads it
  \begin{align*}
  I: \text{sub } r4, r1, r3 \\
  J: \text{add } r1, r2, r3 \\
  K: \text{mul } r6, r1, r7
  \end{align*}

• Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”.

• Can’t happen in MIPS 5 stage pipeline because:
  – All instructions take 5 stages, and
  – Reads are always in stage 2, and
  – Writes are always in stage 5

Three Generic Data Hazards

• Write After Write (WAW)
  Instr\textsubscript{j} writes operand \textit{before} Instr\textsubscript{i} writes it.
  \begin{align*}
  I: \text{sub } r1, r4, r3 \\
  J: \text{add } r1, r2, r3 \\
  K: \text{mul } r6, r1, r7
  \end{align*}

• Called an “output dependence” by compiler writers. This also results from the reuse of name “r1”.

• Can’t happen in MIPS 5 stage pipeline because:
  – All instructions take 5 stages, and
  – Writes are always in stage 5

• Will see WAR and WAW in more complicated pipes

Forwarding to Avoid Data Hazard

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{forwarding_diagram}
\caption{Time (clock cycles) for instructions in order: add \textit{r1}, \textit{r2}, \textit{r3}; sub \textit{r4}, \textit{r1}, \textit{r3}; and \textit{r6}, \textit{r1}, \textit{r7}; or \textit{r8}, \textit{r1}, \textit{r9}; xor \textit{r10}, \textit{r1}, \textit{r11}.}
\end{figure}

HW Change for Forwarding

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{hw_change_diagram}
\caption{What circuit detects and resolves this hazard?}
\end{figure}
Forwarding to Avoid LW-SW Data Hazard

<table>
<thead>
<tr>
<th>Instruction Order</th>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td>Fetch</td>
</tr>
<tr>
<td>lw r4, 0(r1)</td>
<td>Fetch</td>
</tr>
<tr>
<td>sw r4, 12(r1)</td>
<td>Fetch</td>
</tr>
<tr>
<td>or r8, r6, r9</td>
<td>Fetch</td>
</tr>
<tr>
<td>xor r10, r9, r11</td>
<td>Fetch</td>
</tr>
</tbody>
</table>

Data Hazard Even with Forwarding

<table>
<thead>
<tr>
<th>Instruction Order</th>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw r1, 0(r2)</td>
<td>Fetch</td>
</tr>
<tr>
<td>sub r4, r1, r6</td>
<td>Fetch</td>
</tr>
<tr>
<td>and r6, r1, r7</td>
<td>Fetch</td>
</tr>
<tr>
<td>or r8, r1, r9</td>
<td>Fetch</td>
</tr>
</tbody>
</table>

Software Scheduling to Avoid Load Hazards

Try producing fast code for
\[
a = b + c; \quad d = e - f;
\]
assuming a, b, c, d, e, and f in memory.

Slow code:
- LW Rb,b
- LW Rc,c
- ADD Ra,Rb,Rc
- SW a,Ra
- LW Re,e
- LW Rf,f
- SW a,Ra

Fast code:
- LW Rb,b
- LW Rc,c
- ADD Ra,Rb,Rc
- SW a,Ra
- LW Re,e
- LW Rf,f
- SW a,Ra
Control Hazard on Branches
Three Stage Stall

10: beq r1, r3, 36
14: and r2, r3, r5
18: or r6, r1, r7
22: add r8, r1, r9
36: xor r10, r1, r11

What do you do with the 3 instructions in between?
How do you do it?
Where is the "commit"?

Branch Stall Impact

• If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
• Two part solution:
  – Determine branch taken or not sooner, AND
  – Compute taken branch address earlier
• MIPS branch tests if register = 0 or ≠ 0
• MIPS Solution:
  – Move Zero test to ID/RF stage
  – Adder to calculate new PC in ID/RF stage
  – 1 clock cycle penalty for branch versus 3

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
  – 47% MIPS branches not taken on average
  – PC+4 already calculated, so use it to get next instruction
#3: Predict Branch Taken
  – 53% MIPS branches taken on average
  – But haven’t calculated branch target address in MIPS
    » MIPS still incurs 1 cycle branch penalty
    » Other machines: branch target known before outcome
Four Branch Hazard Alternatives

#4: Delayed Branch
- Define branch to take place AFTER a following instruction
  - branch instruction
  - sequential successor₁
  - sequential successor₂
  - .......
  - sequential successorₙ
  - branch target if taken
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Delayed Branch

• Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled

• Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

Pipeline speedup = (Pipeline depth + Branch frequency x Branch penalty) / 1

Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

<table>
<thead>
<tr>
<th>Scheduling Scheme</th>
<th>Branch Penalty</th>
<th>CPI</th>
<th>Speedup v. Unpipelined</th>
<th>Speedup v. Stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>3</td>
<td>1.60</td>
<td>3.1</td>
<td>1.0</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.20</td>
<td>4.2</td>
<td>1.33</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.40</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.10</td>
<td>4.5</td>
<td>1.45</td>
</tr>
</tbody>
</table>
Problems with Pipelining

- **Exception:** An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode
- **Interrupt:** Hardware signal to switch the processor to a new instruction stream
  - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)
- **Problem:** It must appear that the exception or interrupt must appear between 2 instructions (I<sub>i</sub> and I<sub>i+1</sub>)
  - The effect of all instructions up to and including I<sub>i</sub> is totalling complete
  - No effect of any instruction after I<sub>i</sub> can take place
- The interrupt (exception) handler either aborts program or restarts at instruction I<sub>i+1</sub>

Precise Exceptions in Static Pipelines

Key observation: architected state only change in memory and register write stages.

Summary: Control and Pipelining

- Next time: Read Appendix A
- Control VIA State Machines and Microprogramming
- Just overlap tasks; easy if tasks are independent
- Speed Up \( \leq \) Pipeline Depth; if ideal CPI is 1, then:
  \[
  \text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
  \]
- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction
- Exceptions, Interrupts add complexity
- Next time: Read Appendix C, record bugs online!