CSE 502 Graduate Computer Architecture

Lec 3-5 – Performance + Instruction Pipelining Review

Larry Wittie
Computer Science, StonyBrook University

http://www.cs.sunysb.edu/~cse502 and ~lw

Slides adapted from David Patterson, UC-Berkeley cs252-s06
Review from last lecture

• Tracking and extrapolating technology part of architect’s responsibility
• Expect Bandwidth in disks, DRAM, network, and processors to improve by at least as much as the square of the improvement in Latency
• Quantify Cost (vs. Price)
  – IC ∼ f(Area^2) + Learning curve, volume, commodity, price margins
• Quantify dynamic and static power
  – Capacitance x Voltage^2 x frequency, Energy vs. power
• Quantify dependability
  – Reliability (MTTF vs. FIT), Availability (MTTF/(MTTF+MTTR))
Outline

- Review
  - F&P: Benchmarks age, disks fail, singlepoint fail danger
  - 502 Administrivia
  - MIPS – An ISA for Pipelining
  - 5 stage pipelining
  - Structural and Data Hazards
  - Forwarding
  - Branch Schemes
  - Exceptions and Interrupts
  - Conclusion
Fallacies and Pitfalls (1/2)

- **Fallacies** - commonly held misconceptions
  - When discussing a fallacy, we try to give a counterexample.

- **Pitfalls** - easily made mistakes.
  - Often generalizations of principles true in limited context
  - Text shows Fallacies and Pitfalls to help you avoid these errors

- **Fallacy: Benchmarks remain valid indefinitely**
  - Once a benchmark becomes popular, there is tremendous pressure to improve performance by targeted optimizations or by aggressive interpretation of the rules for running the benchmark: “benchmarksmanship.”
  - 70 benchmarks in the 5 SPEC releases to 2000. 70% dropped from the next release because no longer useful

- **Pitfall: A single point of failure**
  - Rule of thumb for fault tolerant systems: make sure that every component is redundant so that no single component failure can bring down the whole system (e.g, power supply)

Lab rule of thumb: “Don’t buy one of anything.”
Fallacies and Pitfalls (2/2)

- Fallacy - Rated MTTF of disks is 1,200,000 hours or \( \approx 140 \) years, so disks practically never fail
- But disk lifetime is 5 years \( \Rightarrow \) replace a disk every 5 years; on average, 28 replacements, so “never” fail
- A better unit: % that fail (1.2M MTTF = 833 FIT)
- Fail over lifetime: if had 1000 disks for 5 years
  \[ = 1000 \times (5 \times 365 \times 24) \times 833 / 10^9 = 36,485,000 / 10^6 = 37 \]
  \[ = 3.7\% \text{ (37/1000)} \text{ fail over 5 yr lifetime (1.2M hr MTTF)} \]
- But this is under pristine conditions
  - little vibration, narrow temperature range \( \Rightarrow \) no power failures
- Real world: 3% to 6% of SCSI drives fail per year
  - 3400 - 6800 FIT or 150,000 to 300,000 hour MTTF [Gray & van Ingen 05]
- 3% to 7% of ATA drives fail per year (Advanced Tech Attachment)
  - 3400 - 8000 FIT or 125,000 to 300,000 hour MTTF [Gray & van Ingen 05]
CSE502: Administrivia

Instructor: Prof Larry Wittie
Office/Lab: 1308 CompSci, lw AT icDOTsunysbDOTedu
Office Hrs: TuTh, 3:45 - 5:15 pm, if door open, or appt.
T. A.: Unlikely
Class: TuTh, 2:20 - 3:40 pm 2120 Comp Sci
Web page: http://www.cs.sunysb.edu/~cse502/
Current reading: Appendix A (back of CAQA4) (Chap 1 was last week)
(New edition this fall 2011 so little 4ed trade-back value)
Outline

• Review
• F&P: Benchmarks age, disks fail, single-points fail
• 502 Administrivia
• MIPS – An ISA for Pipelining
• 5 stage pipelining
• Structural and Data Hazards
• Forwarding
• Branch Schemes
• Exceptions and Interrupts
• Conclusion
A "Typical" RISC ISA

• 32-bit fixed format instruction (only 3 formats: RIJ)
• 32 32-bit GPR (R0 contains zero, DP takes pair)
• 3-address, reg-reg arithmetic instruction
• Single address mode for load/store:
  base + displacement
  – no indirection (since it needs another memory access)
• Simple branch conditions (e.g., single-bit: 0 or not?)
• (Delayed branch - ineffective in deep pipelines, so
  no longer used)

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

Register-Register – R Format – Arithmetic operations

<table>
<thead>
<tr>
<th>31</th>
<th>26-25</th>
<th>21-20</th>
<th>16-15</th>
<th>11-10</th>
<th>6-5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td></td>
<td>Opx</td>
</tr>
</tbody>
</table>

Register-Immediate – I Format – All immediate arithmetic ops

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

Branch – I Format – Moderate relative distance conditional branches

<table>
<thead>
<tr>
<th>31</th>
<th>26-25</th>
<th>21-20</th>
<th>16-15</th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Opx</td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jump / Call – J Format – Long distance jumps

<table>
<thead>
<tr>
<th>31</th>
<th>26-25</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Datapath vs Control

- **Datapath**: Storage, Functional Units, Interconnections 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
Approaching an ISA

- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL (register transfer language) on architected registers and memory
- Given technology constraints, assemble adequate datapath
  - Architected storage mapped to actual storage
  - Function Units (FUs) to do all the required operations
  - Possible additional storage (eg. Internal registers: MAR, MDR, IR, ... {Memory Address Register, Memory Data Register, Instruction Register})
  - Interconnect to move information among registers and function units
- Map each instruction to a sequence of RTL operations
- Collate sequences into symbolic controller state transition diagram (STD)
- Lower symbolic STD to control points
- Implement controller
5 Steps of a (pre-pipelined) MIPS Datapath

Figure A.2, Page A-8

Stages: 1 Instruction Fetch
2 Instr. Decode Reg. Fetch
3 Execute Addr. Calc
4 Memory Access
5 Write Back

RTL Actions: Reg. Transfer Language
IR <= mem[PC]; #stage 1
PC <= PC + 4

Reg[IR_{rd}] <= (Reg[IR_{rs}] \text{ op}_{IR_{op}} Reg[IR_{rt}]) \#op is done in stages 2-5
5-Stage MIPS Datapath (has pipeline latches)

Figure A.3, Page A-9

IR \leftarrow \text{mem}[PC] \; \#1
PC \leftarrow PC + 4
A \leftarrow \text{Reg}[IR_{rs}] \; \#2
B \leftarrow \text{Reg}[IR_{rt}]
\text{rslt} \leftarrow A \; \text{op}_{IRop} \; B \; \#3
\text{WB} \leftarrow \text{rslt} \; \#4
\text{Reg}[IR_{rd}] \leftarrow \text{WB} \; \#5

Stages: 1
- Instruction Fetch
2
- Instr. Decode Reg. Fetch
3
- Execute Addr. Calc
4
- Memory Access
5
- Write Back

IF/ID
ID/EX
MEM/WB

Adder
Address
Memory
ID/ID
IF/ID
ID/EX
EX/MEM
MEM/WB

IRop
WBR
Mem
Data Memory

Next PC
Next SEQ PC
Next SEQ PC
Next SEQ PC

IR <= \text{mem}[PC]; \#1
PC <= PC + 4
A <= \text{Reg}[IR_{rs}]; \#2
B <= \text{Reg}[IR_{rt}]
\text{rslt} <= A \; \text{op}_{IRop} \; B \; \#3
\text{WB} <= \text{rslt} \; \#4
\text{Reg}[IR_{rd}] <= \text{WB} \; \#5
Instruction Set Processor Controller

IR <= mem[PC];
PC <= PC + 4

A <= Reg[IR_{rs}];
B <= Reg[IR_{rt}]

if bop(A,B)
PC <= PC+IR_{im}

br
jmp

PC <= IR_{jaddr}
r <= A op_{IROP} B
r <= A op_{IROP} IR_{im}
r <= A + IR_{im}

WB <= r
WB <= Mem[r]

Reg[IR_{rd}] <= WB
Reg[IR_{rd}] <= WB
Reg[IR_{rd}] <= WB
5-Stage MIPS Datapath (has pipeline latches)

Figure A.3, Page A-9

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

2/8,10,15/2011  CSE502-S11, Lec 03+4+5-perf & pipes
Visualizing Pipelining

Figure A.2, Page A-8

![Diagram showing pipelining over time.]

Time (clock cycles)

Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7
--- | --- | --- | --- | --- | --- | ---
Ifetch | Reg | ALU | DMem | Reg | Ifetch | Reg
Ifetch | Reg | ALU | DMem | Reg | Ifetch | Reg
Ifetch | Reg | ALU | DMem | Reg | Ifetch | Reg
Ifetch | Reg | ALU | DMem | Reg | Ifetch | Reg

Instr Order

Cycle 1  Cycle 2  Cycle 3  Cycle 4  Cycle 5  Cycle 6  Cycle 7

2/8,10,15/2011  CSE502-S11, Lec 03+4+5-perf & pipes  16
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 (having a single person to fold and put clothes away at same time)
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (having a missing sock in a later wash; cannot put away)
  - **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
One Memory Port/Structural Hazards
(Similar to Figure A.5, Page A-15)

Time (clock cycles)

Cycle 1: Ifetch -> Reg
Cycle 2: Reg -> ALU
Cycle 3: ALU -> DMem
Cycle 4: DMem -> Reg
Cycle 5: Reg -> ALU
Cycle 6: ALU -> DMem
Cycle 7: DMem -> Reg

I
nstr 1

Load

Instr 2

Stall

Instr 3

How do you “bubble” the pipe?
Code SpeedUp 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, Ideal 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

Assume loads are 20% of instructions executed

\[
\text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times \left(\frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}}\right) = \text{Pipeline Depth}
\]

\[
\text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1 + 0.2 \times 1} \times \left(\frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{unpipe}}/1.05}\right) = \left(\frac{\text{Pipeline Depth}}{1.20}\right) \times 1.05 \quad \{105/120 = 7/8\}
\]

\[
= 0.875 \times \text{Pipeline Depth}
\]

\[
\frac{\text{SpeedUp}_A}{\text{SpeedUp}_B} = \frac{\text{Pipeline Depth}}{(0.875 \times \text{Pipeline Depth})} = 1.14
\]

- Machine A is 1.14 times faster
Data Hazard on Register R1 (If No Forwarding)

Figure A.6, Page A-16

**Instr. Order**

- **add r1, r2, r3**
- **sub r4, r1, r3**
- **and r6, r1, r7**
- **or r8, r1, r9**
- **xor r10, r1, r11**

**Time (clock cycles)**

No forwarding needed since write reg in 1st half cycle, read reg in 2nd half cycle.
Three Generic Data Hazards

• Read After Write (RAW)
  Instr\textsubscript{J} tries to read operand before Instr\textsubscript{I} writes it

\[ \text{I: add } r1, r2, r3 \]
\[ \text{J: sub } r4, r1, r3 \]

• Caused by a “(True) Dependence” (in compiler nomenclature). This hazard results from an actual need for communicating a new data value.
Three Generic Data Hazards

• **Write After Read (WAR)**
  \( \text{Instr}_j \) writes operand **before** \( \text{Instr}_i \) reads it

\[
\text{I: } \text{sub} \ r4, r1, r3 \\
\text{J: } \text{add} \ r1, r2, r3 \\
\text{K: } \text{mul} \ r6, r1, r7
\]

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

• Cannot happen in MIPS 5 stage pipeline because:
  – All instructions take 5 stages, and  
  – Register reads are always in stage 2, and  
  – Register writes are always in stage 5
Three Generic Data Hazards

• **Write After Write (WAW)**
  Instr$_J$ writes operand *before* Instr$_I$ writes it.

  
  ![Diagram]
  
  I: sub \textit{r1}, \textit{r4}, \textit{r3}
  J: add \textit{r1}, \textit{r2}, \textit{r3}
  K: mul \textit{r6}, \textit{r1}, \textit{r7}

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

  • Cannot happen in MIPS 5 stage pipeline because:
    – All instructions take 5 stages, and
    – Register writes are always in stage 5

  • Will see WAR and WAW in more complicated pipes
Forwarding to Avoid Data Hazard

Figure A.7, Page A-19

Instr. Order

Time (clock cycles)

add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11

Forwarding of ALU outputs needed as ALU inputs 1 & 2 cycles later.

Forwarding of LW MEM outputs to SW MEM or ALU inputs 1 or 2 cycles later.

Need no forwarding since write reg is in 1\textsuperscript{st} half cycle, read reg in 2\textsuperscript{nd} half cycle.
What circuit detects and resolves this hazard?
Forwarding Avoids ALU-ALU & LW-SW Data Hazards

Figure A.8, Page A-20

Time (clock cycles)

Instr. Order

add r1, r2, r3

lw r4, 0(r1)

sw r4, 12(r1)

or r8, r6, r9

xor r10, r9, r11

Reg ALU DMem

I fetch Reg DMem Reg

I fetch Reg DMem Reg

I fetch Reg ALU DMem Reg

I fetch Reg ALU DMem Reg

I fetch Reg ALU DMem Reg
LW-ALU Data Hazard Even with Forwarding

Figure A.9, Page A-21

Time (clock cycles)

Instr. | Order
--- | ---
lw r1, 0(r2) | lw r1, 0(r2)
sub r4, r1, r6 | sub r4, r1, r6
and r6, r1, r7 | and r6, r1, r7
or r8, r1, r9 | or r8, r1, r9

No forwarding needed since write reg in 1st half cycle, read reg in 2nd half cycle.
Data Hazard Even with Forwarding
(Similar to Figure A.10, Page A-21)

How is this hazard detected?

lw r1, 0(r2)
sub r4,r1,r6
and r6,r1,r7
or r8,r1,r9

No forwarding needed since write reg in 1st half cycle, read reg in 2nd half cycle.
Try producing fast code with no stalls for

\[
\begin{align*}
a &= b + c; \\
d &= e - f;
\end{align*}
\]
assuming \(a, b, c, d, e,\) and \(f\) are in memory.

Slow code:

\[
\begin{align*}
&\text{LW} \quad \text{Rb,}b \\
&\text{LW} \quad \text{Rc,}c \\
&\text{ADD} \quad \text{Ra,}Rb,\text{Rc} \\
&\text{SW} \quad a,\text{Ra} \\
&\text{LW} \quad \text{Re,}e \\
&\text{LW} \quad \text{Rf,}f \\
&\text{SUB} \quad \text{Rd,}Re,\text{Rf} \\
&\text{SW} \quad d,\text{Rd}
\end{align*}
\]

Fast code (no stalls):

\[
\begin{align*}
&\text{LW} \quad \text{Rb,}b \\
&\text{LW} \quad \text{Rc,}c \\
&\text{ADD} \quad \text{Ra,}Rb,\text{Rc} \\
&\text{LW} \quad \text{Re,}e \\
&\text{LW} \quad \text{Rf,}f \\
&\text{SW} \quad a,\text{Ra} \\
&\text{SUB} \quad \text{Rd,}Re,\text{Rf} \\
&\text{SW} \quad d,\text{Rd}
\end{align*}
\]

Compiler optimizes for performance. Hardware checks for safety.

Important technique!
Outline

• Review
• F&P: Benchmarks age, disks fail, single-points fail
• 502 Administrivia
• MIPS – An ISA for Pipelining
• 5 stage pipelining
• Structural and Data Hazards
• Forwarding
• Branch Schemes
• Exceptions and Interrupts
• Conclusion
5-Stage MIPS Datapath (has pipeline latches)

Figure A.3, Page A-9

Stages: 1 Instruction Fetch
   2 Instr. Decode Reg. Fetch
   3 Execute Addr. Calc
   4 Memory Access
   5 Write Back

• Old simple design put branch completion in stage 4 (Mem)

Will move red circuits to 2nd stage to make branch delays shorter

2/8,10,15/2011 CSE502-S11, Lec 03+4+5-perf & pipes
Control Hazard on Branch - Three Cycle Stall
(Caused if Decide Branches in 4th Stage)

10: beq r1, r3, 34
14: and r2, r3, r5
18: or r6, r1, r7
22: add r8, r1, r9
34: xor r10, r1, r11

What can be done with the 3 instructions between beq & xor?
Code between beq&xor must not start until know beq not branch => 3 stalls
Adding 3 cycle stall after every branch (1/7 of instructions) => CPI += 3/7. BAD!
Branch Stall Impact if Commit in Stage 4

- If CPI = 1 and 15% of instructions are branches, Stall 3 cycles => new CPI = 1.45 \( (1+3\times0.15) \) Too much!

- Two-part solution:
  - Determine sooner whether branch taken or not, AND
  - Compute taken branch address earlier

- MIPS branch tests if register = 0 or ≠ 0

- Original 1986 MIPS Solution:
  - Move zero_test to ID/RF (Inst Decode & Register Fetch) stage(2) \( (4=MEM) \)
  - Add extra adder to calculate new PC (Program Counter) in ID/RF stage
  - Result is 1 clock cycle penalty for branch versus 3 when decided in MEM
The fast_branch design needs a slightly longer stage 2 cycle time, making the clock a little slower for all stages.

- Example of interplay of instruction set design and cycle time.
Four Branch Hazard Alternatives

#1: Stall until branch direction is clearly known
#2: Predict Branch Not Taken – Easy Solution
   - Execute the next instructions in sequence
   - PC+4 already calculated, so use it to get next instruction
   - Nullify bad instructions in pipeline if branch is actually taken
   - Nullify easier since pipeline state updates are late (MEM, WB)
   - 47% MIPS branches not taken on average

#3: Predict Branch Taken
   - 53% MIPS branches taken on average
   - But have not calculated branch target address in MIPS
     » MIPS still incurs 1 cycle branch penalty
     » Some other CPUs: branch target known before outcome
Last of Four Branch Hazard Alternatives

#4: Delayed Branch (Used Only in 1st MIPS “Killer Micro”)
- Define branch to take place AFTER a following instruction

  \[
  \text{branch instruction} \quad \text{sequential successor}_1 \quad \text{sequential successor}_2 \quad \ldots \ldots \quad \text{sequential successor}_n \quad \text{branch target if taken}
  \]

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS 1st used this (Later versions of MIPS did not; pipeline deeper)
Scheduling Branch Delay Slots (Fig A.14)

A. From before branch

- add $1,$2,$3
- if $2=0$ then
- delay slot
- becomes
- if $2=0$ then
- add $1,$2,$3

B. From branch target

- sub $4,$5,$6
- add $1,$2,$3
- if $1=0$ then
- delay slot
- becomes
- sub $4,$5,$6
- add $1,$2,$3
- if $1=0$ then
- sub $4,$5,$6

C. From fall through

- add $1,$2,$3
- if $1=0$ then
- delay slot
- becomes
- add $1,$2,$3
- if $1=0$ then
- sub $4,$5,$6
- sub $4,$5,$6

- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the $\text{sub}$ instruction may need to be copied, increasing IC
- In B and C, must be okay to execute an extra $\text{sub}$ when branch fails
Delayed Branch Not Used in Modern CPUs

• Compiler effectiveness 1/2 for single branch delay slot:
  – Fills about 60% of branch delay slots
  – About 80% of instructions executed in branch delay slots useful in computation
  – Only half of (60% x 80%) slots usefully filled; cannot fill 2 or more

• Delayed Branch downside: As processor designs use deeper pipelines and multiple issue, the branch delay grows and needs many more delay slots
  – Delayed branching soon lost effectiveness and popularity compared to more expensive but more flexible dynamic approaches
  – Growth in available transistors soon permitted dynamic approaches that keep records of branch locations, taken/not-taken decisions, and target addresses
  – Multi-issue 2 => 3 delay slots needed, 4 => 7 slots, 8 => 15 slots
Evaluating Branch Alternatives

\[
\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}
\]

Assume 4% unconditional jump, 10% conditional branch-taken, 6% conditional branch-not-taken, base CPI = 1.

<table>
<thead>
<tr>
<th>Scheduling Scheme</th>
<th>Branch penalty</th>
<th>CPI</th>
<th>speedup vs. no-pipe 5 cycles</th>
<th>speedup vs. stall_pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline (Stage 4)</td>
<td>3</td>
<td>1.60</td>
<td>3.1</td>
<td>1.00</td>
</tr>
<tr>
<td>Predict taken (Stage 2)</td>
<td>1</td>
<td>1.20</td>
<td>4.2</td>
<td>1.33</td>
</tr>
<tr>
<td>Predict not taken (St.2)</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.40</td>
</tr>
<tr>
<td>Delayed branch (Stg 2)</td>
<td>0.5</td>
<td>1.10</td>
<td>4.5</td>
<td>1.45</td>
</tr>
</tbody>
</table>

(Sample 1.60 = 1 + 3(4 + 10 + 6)%  (4.5 = 5/1.10)  (1.45 = 1.6/1.1)

Calculations) 1.20 = 1 + 1(4 + 10 + 6)%  (to calculate taken target)

(1.14 = 1 + 1(4 + 10)%  (refetch for jump, taken-branch)
Another Problem with Pipelining

- **Exception**: An unusual event happens to an instruction during its execution {caused by instructions executing}
  - Examples: divide by zero, undefined opcode

- **Interrupt**: Hardware signal to switch the processor to a new instruction stream {not directly caused by code}
  - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)

- **Precise Interrupt Problem**: Must seem as if the exception or interrupt appeared between 2 instructions ($I_i$ and $I_{i+1}$) although several instructions were executing at the time
  - All instructions up to and including $I_i$ are totally completed
  - No effect of any instruction after $I_i$ is allowed to be saved

- After a precise interrupt, the interrupt (exception) handler either aborts the program or restarts at instruction $I_{i+1}$
Precise Exceptions in Static Pipelines

Key observation: “Architected” states change only in memory (M) and register write (W) stages.
And In Conclusion: Control and Pipelining

- Quantify and summarize performance
  - Ratios, Geometric Mean, Multiplicative Standard Deviation
- F&P: Benchmarks age, disks fail, single-point failure
- Control via State Machines and Microprogramming
- Just overlap tasks; easy if tasks are independent
- Speed Up ≤ 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 by stalling:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch or branch (taken/not-taken) prediction
- Exceptions and interrupts add complexity

- For next time: Read Appendix C.
CSE502: Administrivia

http://www.cs.sunysb.edu/~lw/teaching/cse502/DoldF07/

Last year's slides are in ~lw/teaching/cse502/DoldF08/
DoldF07/lec01-intro.pdf
DoldF07/lec02-intro.pdf
DoldF07/lec03-pipe.pdf
DoldF07/lec04-cache.pdf
DoldF07/lec05-dynamic-sched.pdf
DoldF07/lec06-dynamic-schedB.pdf
DoldF07/lec07-ILP limits.pdf
DoldF07/lec07-limitsILP_SMT.pdf
DoldF07/lec08-SMT.pdf
DoldF07/lec09-Vector.pdf
DoldF07/lec10-Modern Vector.pdf
DoldF07/lec11-SMP.pdf
DoldF07/lec12-Snoop+MTreview.pdf
DoldF07/lec12-Snoop+MTreviewPreliminary.pdf
DoldF07/lec14-directory.pdf
DoldF07/lec16-T1 MP.pdf
DoldF07/lec17-memoryhier.pdf
DoldF07/lec18-VM memhier2.pdf
DoldF07/lec19-storage.pdf
DoldF07/lec20-review.pdf