Computer Systems Architecture
CMSC 411
Unit 3 – Instruction Pipelining

Alan Sussman
September 21, 2004

So far

• What we mean by computer performance
• How to measure it
• How instruction sets are designed
• How the design influences performance
What’s next

- A variety of hardware and compiler techniques to speed the execution of programs
  - What is pipelining? (Section A.1)
  - How does MIPS divide instructions into stages or cycles? (A.1)
  - What kinds of overheads are there in pipelining? (A.1)
  - How much speedup do we get? (A.1)
  - What are structural hazards, data hazards, and control hazards? (A.2)
  - How are these techniques used to reduce stalls:
    - data forwarding? (A.2)
    - instruction reordering? (A.2)
    - compiler approaches to reduce branch delays? (A.2)

What is pipelining?

- **Pipelining** is an implementation technique whereby multiple instructions are overlapped in execution
- In other words, at any given moment in the execution of a computer program, many different instructions are at various stages of completion!
- Example: Car wash
Throughput

• The number of instructions that complete per unit time
• Instructions take many clock cycles
  Ideally, every clock cycle, we want a new instruction to begin (and end)
• This is how we will improve throughput

A MIPS implementation without pipelining

• Recall from CMSC 311 that instructions execute in different stages or cycles
  – Instruction fetch cycle (IF): fetch the instruction from memory and update the program counter (PC) to point to the next instruction. Note: We’re not using the NPC register that the book introduces.
    
    \[
    \text{IR} \leftarrow \text{Mem}[\text{PC}]
    \]
    \[
    \text{PC} \leftarrow \text{PC} + 4
    \]
MIPS w/o pipelining (cont.)

- **Instruction decode cycle (ID)**: Put the operands in pipeline registers $A$ and $B$. Sign-extend the low order 16 bits of the IR and store in pipeline register $Imm$. (This sometimes holds the "immediate" constant.)

  \[
  \begin{align*}
  A & \leftarrow \text{Regs}[\text{IR}_{6..10}] \\
  B & \leftarrow \text{Regs}[\text{IR}_{11..15}] \\
  Imm & \leftarrow (\text{IR}_{16})^{16##\text{IR}_{16..31}} 
  \end{align*}
  \]

MIPS w/o pipelining (cont.)

- **Execution cycle (EX)**: Use the ALU
- If memory reference:
  \[
  \text{ALUOutput} \leftarrow A + \text{Imm}
  \]
- If register-register ALU instruction:
  \[
  \text{ALUOutput} \leftarrow A \text{ op } B
  \]
- If register-immediate ALU instruction:
  \[
  \text{ALUOutput} \leftarrow A \text{ op } \text{Imm}
  \]
- If branch instruction: compute the branch address and check the branch condition:
  \[
  \begin{align*}
  \text{ALUOutput} & \leftarrow \text{PC} + (\text{Imm} \ll 2) \\
  \text{Cond} & \leftarrow (A \text{ op } 0)
  \end{align*}
  \]

(but $PC$ or $Imm$ should be adjusted down by 4 to make this work right).
MIPS w/o pipelining (cont.)

- **Memory access cycle (MEM)**: finish loads, stores, and branches:
  - Load: $\text{LMD} \leftarrow \text{Mem}[^{\text{ALUOutput}}]$
  - Store: $\text{Mem}[^{\text{ALUOutput}}] \leftarrow B$
  - Branch: if $\text{Cond}$ then $\text{PC} \leftarrow \text{ALUOutput}$
    
    else $\text{PC}$ is ok

MIPS w/o pipelining (cont.)

- **Write-back cycle (WB)**: update the registers
  - Register-register ALU instruction:
    $\text{Regs}[^{\text{IR}_{16..20}}] \leftarrow \text{ALUOutput}$
  - Register-immediate ALU instruction:
    $\text{Regs}[^{\text{IR}_{11..15}}] \leftarrow \text{ALUOutput}$
  - Load instruction:
    $\text{Regs}[^{\text{IR}_{11..15}}] \leftarrow \text{LMD}$
Some notes

- All instructions take 4-5 cycles
- Some of the temporary registers could be eliminated, but they become convenient in a minute for pipelining
- Could build the architecture so that these 5 cycles are one clock cycle, not 5
- We assume that there are separate data paths for instruction memory and for data memory (so loads and stores don’t interfere with instruction fetch), usually implemented via separate caches

A pipelined implementation of MIPS

Suppose we have 5 load/store instructions to execute, all involving different registers and memory locations. Fig. A.1

<table>
<thead>
<tr>
<th>Inst. #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
With pipeline registers

![Diagram of pipeline registers]

Figure A.3

Communication paths and timing

![Diagram of communication paths and timing]

Figure A.17
Ideal throughput from pipelining

• Example 1: Suppose we have 100 load/store instructions to execute, and we arrange to have no register conflicts
• Time for original MIPS implementation: 100 instructions × 5 cycles per instruction = 500 cycles
• Time for pipelined MIPS implementation: 1st instruction takes 5 cycles. The others each finish 1 cycle later than the preceding one.
  Time = 5 + 99 = 104 cycles
• Speedup = 500/104 ≈ 5
• This is the “ideal case” - life is not that simple

A more realistic case

• Example 2: Suppose that in our original MIPS implementation, we could run the stages this fast:
  – IF - 10ns
  – ID - 8ns
  – EX - 7ns
  – MEM - 10ns
  – WB - 5ns
• Then time for original MIPS implementation of 100 load/store instructions:
  – 100 instructions × 40ns per instruction = 4000 ns
Example 2 (cont.)

- Time for pipelined MIPS implementation:
  We have to synchronize the stages, so we need to run the clock at 10 ns
- 1st instruction takes 50 ns. The others each finish 1 cycle later than the preceding one.
  \[ \text{Time} = 50 \text{ ns} + 99 \times 10 \text{ ns} = 1040 \text{ ns} \]
- Speedup = \( \frac{4000}{1040} \approx 3.85 \)

Even more realistic case

- Example 3: The original MIPS implementation doesn't always need to use the MEM cycle
  - IF - 10 ns
  - ID - 8 ns
  - EX - 7 ns
  - MEM - 10 ns
  - WB - 5 ns
- Suppose that only 30% of instructions use memory access. So, on average, for every 100 instructions, we have about 70 that use 4 stages and 30 that use 5.
Example 3 (cont.)

• Time for original MIPS implementation:
  – 70 instructions × 30 ns per instruction +
    30 instructions × 40 ns per instruction =
    3300 ns

• Time for pipelined MIPS implementation: We
  have to synchronize the stages, so we need to run
  the clock at 10 ns, and we need 5 cycles for every
  instruction.
  – 1st instruction takes 50 ns. The others each finish 1
    cycle later than the preceding one
  – Time = 50 ns + 99*10 ns = 1040 ns

• Speedup = 3300/1040 ≈ 3.17

Overhead of pipelining

• We just summarized the two major overhead costs
  in pipelining:
  – making the time for every stage equal the time for the
    longest stage
  – making the time for every instruction equal the time for
    the longest instruction (not quite true, but true for a
    wide range of instructions)

• Unfortunately, the speedup of pipelining is
  reduced even further by hazards that cause
  “bubbles” in the pipeline
Pipeline hazards cause stalls

- When some instruction is unable to complete on schedule, we must
  - finish the earlier instructions on schedule
  - delay the later instructions
- This is called stalling the pipeline

Pipeline hazards

- What causes delays in instruction completion?
  - **Structural hazards** are hardware delays
    Example: memory does not respond to a request as fast as it is expected to
  - **Data hazards** arise when data can be predicted to be unready at the time it is needed
    Example: an instruction needs a register that a previous instruction is still modifying
  - **Control hazards** arise when we need to do something other than incrementing the PC by 4
    Example: conditional branch, jump
Pipeline hazards (cont.)

Pipeline hazards reduce throughput and speedup even more! Fig. A.5

Structural hazard – a load with 1 memory port for data/instructions

Clock cycle

<table>
<thead>
<tr>
<th>Inst #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CMSC 411 - Alan Sussman

Computer Systems Architecture
CMSC 411
Unit 3 – Instruction Pipelining

Alan Sussman
September 23, 2004
Administrivia

- Quiz #1 Tuesday
  - on Units 1 & 2
- If you want to talk to me before quiz, make an appointment, by email
- Read Appendix A for Unit 3

Last time

- MIPS64
  - floating point and control instructions
  - questions?
- Pipelining
  - to overlap instructions in time
  - goal is to complete/start a new instruction every clock cycle
  - instruction goes through stages
    - IF – get the instruction, increment PC
    - ID – read registers, decode rest of instruction
    - EX – use ALU (arithmetic op, address computation, branch address computation)
    - MEM – memory access for load/store
    - WB – write register
Last time (cont.)

- Pipeline overheads include
  - every stage has to take the same time (only 1 clock)
  - every instruction has to go through all stages
- Hazard further reduce pipelining speedup by causing stalls
  - structural hazards are hardware delays
  - Data hazards from data not being ready when it is needed
  - control hazards from control flow changes (jump, branch, etc.)

MIPS pipeline

Figure A.3
Communication paths and timing

Figure A.17

Pipeline hazards (cont.)

- Example 4: In Example 3, had on average, 70 instructions that use 4 stages and 30 that use 5
- Time for original MIPS implementation = 3300 ns
- Suppose that 5 of those instructions involve branches. So 5 times, need to wait until the ID cycle of one instruction is complete before start the IF cycle of the next instruction
  - assume that address calculated in ID (not EX) stage using a separate adder
- Therefore, the next instruction will start 2 cycles later, not 1. So add 5 cycles to the time.
Example 4 (cont.)

- Time for pipelined MIPS implementation:
  - 1st instruction takes 50 ns. The others each finish 1 cycle later than the preceding one, but there is a 5 cycle hazard penalty
  - Time = 50 ns + 99*10 ns + 5*10 ns = 1090 ns

- Speedup = $\frac{3300}{1090} \approx 3.03$

Data hazards

- A **data hazard** occurs when a piece of data is not available when it is needed
  - Perhaps there was a *cache miss*: we expected the value to be in cache, but instead we need to find it in memory
  - Perhaps it is involved in a previous computation that has not yet completed
Types of data hazards

- **RAW**: read after write
  - One instruction writes a value
  - A later instruction reads it
  - Problem: an old value may be read

- **WAW**: write after write
  - One instruction writes a value
  - A later instruction writes in the same location
  - Problem: the final value may be the first, rather than the second

- **WAR**: write after read
  - One instruction reads a value
  - A later instruction writes in the same location
  - Problem: the value read may be the changed value rather than the original; this ordinarily cannot happen
How to avoid data hazard stalls: forwarding

- Need to move data more quickly from the ALU output to the ALU inputs
- So forward the output by designing the circuits so that the ALU output is always fed back (immediately) into the ALU input latches (pipeline registers), adding a circuit to choose between the ALU output and the other registers (a multiplexer)
Sometimes forwarding not enough

*Example:* Data needs to be loaded from memory at least two instructions before use in order to avoid a stall – Figure A.9

Forwarding (cont.)

*Example:*
- \( a = b + c + d \);
- \( e = d - f \);

*Compilers need to be smart enough to prevent stalls when possible*

- **LD** R1, b
- **LD** R2, c
- **LD** R3, d \sloppy ADD can’t be done yet
- **DADD** R4,R1,R2
- **DADD** R4,R3,R4 \ ok by forwarding
- **LD** R5, f \ need to start this before \( a = b + c + d \) completes
- **SD** a, R4
- **DSUB** R6,R3,R5
- **SD** e, R6 \ ok by forwarding
Forwarding (cont.)

• Rules for interchanging instructions:
  – must be in same block (i.e., no branches between them)
  – must check graph of dependencies to make sure they are independent

How the MIPS pipeline introduces stalls

• Data hazards are checked during instruction decode (ID) - if a hazard exists, the EX cycle is delayed (i.e., the instruction is not issued); a "no-op" is issued instead
• The ID cycle also determines whether data forwarding is needed
Control hazards

- **Question**: When do we find out that the PC needs to be modified?
- **Answer**: In pipeline stage ID of a branch instruction
- So, if a branch is taken (i.e., if the PC is modified), then have to wait until the *next* cycle before fetching the correct instruction

Control hazards (cont.)

<table>
<thead>
<tr>
<th>Branch inst.</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch successor</td>
<td>IF</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>Successor + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>Successor + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Wastes 1 *clock cycle*
Example

• If branch in 30% of instructions, then instead of executing 1 instruction per cycle, have 70% of instructions executing in 1 cycle and 30% of instructions executing in 2 cycles
• An average of .7 + .6 = 1.3 cycles per instruction
  – Worse by 30%

Compiler approaches to branch delays

• **Freeze** or **flush** the pipeline when determine that a branch is taken - refer back to Figure A.11 (a stall is inserted)
• **Predict not taken**: continue to begin execution of instructions as if the branch is not taken, but change them to a "no-op" if the branch is taken
Predict not taken scheme – Fig. A.12

<table>
<thead>
<tr>
<th>Untaken branch</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst. i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst. i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst. i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst. i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Taken branch</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst. i+1</td>
<td>IF</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>B.t. + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>B.t. + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Administrivia

• Quiz today, on Units 1 & 2
  – questions?
• Keep reading Appendix A for Unit 3
• HW #3 posted soon

Last time

• Data hazards
  – data needed for an operation is not yet available
  – e.g., cache miss, pipeline has not yet produced the desired value
  – RAW, WAW, WAR
  – forwarding takes data from pipeline registers, sends it to ALU inputs (or another pipeline stage inputs), bypassing registers
  – compiler can reorder instructions to minimize effects of data hazards (stalls/bubbles)
Last time (cont.)

- Control hazards
  - from control flow changes – branches, jumps
  - from not knowing branch direction (or target address) until after ID stage of pipeline
  - prediction strategies allow for hazards causing bubbles only when prediction is wrong
    - e.g., predict not taken scheme causes bubble only if branch is taken (wrong instruction is turned into a no-op)

Branch delays (cont.)

- **Predict taken**: Good if most of the branches are from loops
- Schedule using **branch delay slots**, reordering the code to test the branch earlier
  - always execute the instruction after the branch instruction
  - so can correctly fetch either the instruction after the branch delay slot, or the branch target, knowing whether the branch was taken
Scheduling branch delay slot

- If taken from before branch
  - branch must not depend on rescheduled instruction
  - always improves performance
- If taken from branch target
  - must be OK to execute rescheduled instructions if branch not taken, and may need to duplicate insts.
  - performance improved when branch taken
- If taken from fall through
  - must be OK to execute insts. if branch taken
  - improves performance when branch not taken
Compiler approaches (cont.)

- Some machines have **cancelling** branches, to change the next instruction to a "no-op" when necessary - then there are no requirements on the scheduling strategy.
- In a set of benchmarks not shown in book, 70% of the branch hazards in simpler version of MIPS can be eliminated by branch scheduling.

Summary

- Pipelining can speed instruction execution
- But need to deal with structural hazards, data hazards, and control hazards
- Next
  - How to handle exceptions?
  - How to handle long instructions, such as floating point arithmetic?
The problem

• Question: What makes pipelining hard to implement?
• Answer: Surprises
• Technical names for surprises:
  – exceptions
  – faults
  – interrupts

Some examples of exceptions

• Request for I/O
• Arithmetic troubles: overflow or underflow
• Page fault: data not in (physical) memory
• Illegal address, giving a memory protection violation
• Hardware failure
Classifying exceptions

- **Synchronous**: repeatable every time
  Example: DIV R2, R2, R0
- **Asynchronous**: caused by external events like hardware failure and devices external to processor and memory
- **User requested**: user task asks for it (example: breakpoint)
  **Coerced**: cannot be predicted by user
- **User maskable**: can be disabled by user task
  Example: arithmetic exception
- **Nonmaskable**: cannot be turned off
  Example: hardware failure

Classifying exceptions (cont.)

- **Within instruction**: prevents instruction from completing
  **Between instructions**: no instruction prevented
- **Terminating**: stops the task
  **Resuming**: task can continue

- Machines that handle exceptions, save the state, and then restart correctly are said to be **restartable**
### Categorizing exceptions – Fig. A. 27

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Synch. vs. asynch.</th>
<th>User request vs. coerce</th>
<th>User maskable vs. not</th>
<th>Within vs. between instructions</th>
<th>Resume vs. terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device request</td>
<td>Asynch</td>
<td>Coerced</td>
<td>Not</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Invoke OS</td>
<td>Synch</td>
<td>User req.</td>
<td>Not</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Tracing instructions</td>
<td>Synch</td>
<td>User req.</td>
<td>Maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Synch</td>
<td>User req.</td>
<td>Maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Integer overflow</td>
<td>Synch</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Floating pt. overflow/ underflow</td>
<td>Synch</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
</tbody>
</table>

### Categorizing exceptions (cont.)

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Synch. vs. asynch.</th>
<th>User request vs. coerce</th>
<th>User maskable vs. not</th>
<th>Within vs. between instructions</th>
<th>Resume vs. terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page fault</td>
<td>Synch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Misaligned memory access</td>
<td>Synch</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Mem. prot. violation</td>
<td>Synch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>Synch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Hardware malfunction</td>
<td>Asynch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Power failure</td>
<td>Asynch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Terminate</td>
</tr>
</tbody>
</table>
Computer Systems Architecture  
CMSC 411  
Unit 3 – Instruction Pipelining  

Alan Sussman  
September 30, 2004  

---  

**Administrivia**  

- Quiz 1 returned today  
  - answers posted  
  - Average: 64  
  - Median: 62  
    - 25%: 48  
    - 75%: 80  
- HW #3 posted  
  - due Oct. 7
Last time

- Dealing with control hazards
  - predict taken or not taken
    - cancelling branches variant of this
  - branch delay slot
    - take instruction from before, from target, or from fall-through
- Handling exceptions in the pipeline
  - synchronous vs. asynchronous
  - user requested vs. coerced
  - maskable vs. non-maskable
  - within vs. between instructions
  - terminating vs. resumable

The most difficult exceptions...

- ... are those that occur within EX or MEM stages and need to be handled in a restartable way
- Why difficult? Handling one includes:
  - the next IF gets a "trap instruction"
  - until the trap is taken, turn off all "writes" for the faulting instruction and those that follow it
  - what does the trap do?
    - The trap transfers control to the exception handling routine in the operating system, which saves the PC of the faulting instruction and handles the fault
    - the task is then resumed, using the saved PC and the MIPS instruction RFE or something like it
- Note: May need to save several PCs if delayed branches are involved
Exceptions (cont.)

- Ideally, pipeline can be interrupted so that instructions before the fault complete. Then want to restart execution just after the faulting instruction - **precise exception handling**
- This is the right way to do it, but sometimes architects/manufacturers take shortcuts

When do MIPS exceptions occur?

- **IF**
  - page fault on instruction fetch
  - misaligned memory access
  - memory protection violation
- **ID**
  - undefined or illegal opcode
- **EX**
  - arithmetic exception
- **MEM**
  - page fault on data fetch/store
  - misaligned memory access
  - memory protection violation
- **WB**: None!
Examples of exception handling

<table>
<thead>
<tr>
<th>LD</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

- Handle the MEM fault first, then restart

<table>
<thead>
<tr>
<th>LD</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

- IF fault occurs first, even though LD will fault later
- But for precise exceptions, must handle LD fault first

How is this done?

- **Answer:** Don't handle exceptions until the WB stage
  - each instruction has an associated status vector that keeps track of faults
  - any bit set in the status vector turns off register writes and memory writes
  - in WB stage, the status vector is checked and any fault is handled
  - So, since instructions reach WB in proper order, faults for earlier instructions are handled *before* faults for later instructions
    - Unfortunately, will need to violate this later (for instructions that don’t reach WB in proper order)
Commitment

- When an instruction is guaranteed to complete, it is **committed**
- Life is easier if no instruction changes the permanent machine state before it is committed
- In MIPS, commitment occurs at the end of the MEM stage - that’s why register update occurs in the stage after that
- Some machines muddy the state before commitment, and the exception handler must do its best to restore the state that existed before the instruction started

Complications caused by long instructions

- So far, all MIPS instructions take 5 cycles
- But haven't talked yet about the floating point instructions
- Take it on faith that floating point instructions are inherently slower than integer arithmetic instructions
  - doubters may consult Appendix H in H&P online
How slow is slow?

- Some typical times:
  - **latency** is the number of cycles between an instruction that produces a result and one that uses it
  - **initiation interval** is the number of cycles between two instructions of the same kind (for example, two ADD.Fs)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Initiation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU uses</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Load/store</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADD.F, SUB.F</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>DIV.F</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>

Examples

- If have a string of instructions:
  - DADD
  - DSUB
  - AND
  - OR
  - SLLI

- then there are no delays in the pipeline, because initiation=1 means can start one of these instructions every cycle, and latency=0 means that results from one instruction will be available when the next instruction needs them.
Examples (cont.)

- Suppose have a string of instructions
  - ADD.F
  - SUB.F
- Then initiation=1 means that can start SUB.F one cycle behind ADD.F
- But latency=3 means that this will work right only if SUB.F doesn't need ADD.F's results
- If it does need the results, then need two instructions in between ADD.F and SUB.F to prevent bubbles in the pipeline
Examples (cont.) - Fig. A.32

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL.D</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>IF</td>
<td>ID</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Italic* shows where data is needed, *bold* where a result is available

Hazards caused by long instructions

- The floating point adder and multiplier are pipelined, but the divider is not - that is why the initiation interval for divide is 25
  - A program will run very slowly if it does too many of these!
- It will also run slowly if the results of the divide are needed too soon
Computer Systems Architecture
CMSC 411
Unit 3 – Instruction Pipelining

Alan Sussman
October 5, 2004

Administrivia

• HW #3 due Thursday
• Quiz 2 next Tuesday, Oct. 12
  – on Unit 3, basic pipelining
  – practice quiz posted very soon, answers after class Thursday
• Start reading Chapter 5 on memory hierarchy
  – 5.11-5.15 are not required, but may help
Last time

• Precise exception handling
  – so an instruction that faults can restart correctly
  – turn next instruction into a “trap”, turn off writes for faulting inst. and all after, transfer control to OS exception handler
  – on completion of handler, resume at faulting instruction using RFE, or something like that

• Deal with exceptions in WB pipeline stage
  – to allow handling exceptions in the same order instructions are issued – works for simple MIPS pipeline
  – because that is when MIPS instructions commit

Last time (cont.)

• Long instructions
  – latency is when a result can be used
  – initiation interval is how frequently instruction can be issued to same functional unit
  – with different latencies for different units, instructions can complete out-of-order
  – stalls may be needed if results are not ready – and they may be hard for a compiler to fill
Functional units - Fig. A.31

FP stalls from RAW hazards – Fig. A.33

<table>
<thead>
<tr>
<th>Inst.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F4,0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL.D F0,F4,F6</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td></td>
</tr>
<tr>
<td>ADD.D F2,F0,F8</td>
<td>IF</td>
<td>stall</td>
<td>ID</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F2,0(R2)</td>
<td>stall</td>
<td>IF</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inst.</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL.D</td>
<td>M6</td>
<td>M7</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>stall</td>
<td>stall</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>S.D</td>
<td>stall</td>
<td>stall</td>
<td>ID</td>
<td>EX</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>MEM</td>
</tr>
</tbody>
</table>
Long instructions (cont.)

- It is possible that two instructions enter the WB stage at the same time

<table>
<thead>
<tr>
<th></th>
<th>ADD.D</th>
<th>LD</th>
<th>DADD</th>
<th>DADD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
</tr>
<tr>
<td></td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>A1</td>
<td>ALU</td>
<td>ALU</td>
<td>ALU</td>
<td>ALU</td>
</tr>
<tr>
<td>A2</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- A structural hazard

Long instructions (cont.)

- Instructions can finish in the wrong order
- This can cause WAW hazards
  – see p. A-52 of H&P for an example
- This violation of WB ordering defeats the previous strategy for precise exception handling
**WAW structural hazard – Fig. A.34**

<table>
<thead>
<tr>
<th>MUL,D F0,F4,F6</th>
<th>IF</th>
<th>ID</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD,D F2,F4,F6</td>
<td>IF</td>
<td>ID</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F2,0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**How to detect hazards in ID**

- Early detection would prevent trouble
- **Check for structural hazards:**
  - will the divide unit clear in time?
  - will WB be possible when we need it?
- **Check for RAW data hazards:**
  - will all source registers be available when needed?
- **Check for WAW data hazards:**
  - Is the destination register for any ADD.D, multiply or divide instruction the same register as the destination for this instruction?
- If anything dangerous could happen, delay the execute cycle so no conflict occurs
Precise exception handling for long instructions

Example:

```plaintext
DIV.D   F0, F2, F4
ADD.D   F10, F10, F8
SUB.D   F12, F12, F14
```

• Suppose
  – ADD.D completes,
  – then SUB.D has a floating-point exception,
  – then DIV.D detects an exception

• Big trouble, because ADD.D has destroyed register F10

Possible fixes

• Give up and just do **imprecise exception handling**
  – tempting, but very annoying to users

• **Delay WB** until all previous instructions complete
  – since so many instructions can be active, this is expensive - requires a lot of supporting hardware

• Write, to memory, a **history file** of register and memory changes so can undo instructions if necessary
  – or keep a **future file** of computed results that are waiting for MEM or WB
Possible fixes (cont.)

• Let the **exception handler** finish the instructions in the pipeline and then restart the pipe at the next instruction

• Have the floating point units **diagnose exceptions in their first or second stages**, so can handle them by methods that work well for handling integer exceptions

A case study: MIPS R4000 pipeline design

• MIPS64 architecture, with deeper 8 stage pipeline
  – to get higher clock rates
  – extra stages come from memory accesses
  – techniques called **superpipelining**
MIPS R4000 pipeline stages

- **IF** – 1
  - st half instruction fetch
  - PC selection and start instruction cache access
- **IS** – 2
  - nd half instruction fetch
  - complete instruction cache access
- **RF** – instruction decode, register fetch, hazard checking, instruction cache hit detection
- **EX** – execution
  - includes effective address computation, ALU operation, branch target computation and condition evaluation

MIPS R4000 pipeline (cont.)

- **DF** – 1
  - st half data fetch
  - 1
  - nd half of data cache access
- **DS** – 2
  - nd half data fetch
  - complete data cache access
- **TC** – tag check
  - determine whether data cache access *hit*
- **WB** – write back for loads and ALU operations
MIPS R4000 pipeline (cont.)

A 2 cycle load delay – Fig. A.38

- Time (in clock cycles)
- LD R1
  - Instruction memory
  - Data memory
- Instruction 1
  - Instruction memory
  - Data memory
- Instruction 2
  - Instruction memory
  - Data memory
- ADD R2, R1
  - Instruction memory
  - Data memory

© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman

MIPS R4000 pipeline (cont.)

A 3 cycle branch delay – 1 delay slot + 2 cycle stall for taken branch (untaken just delay slot)

- Time (in clock cycles)
- BEQZ
  - Instruction memory
  - Data memory
- Instruction 1
  - Instruction memory
  - Data memory
- Instruction 2
  - Instruction memory
  - Data memory
- Instruction 3
  - Instruction memory
  - Data memory
- Target
  - Instruction memory
  - Data memory

© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman
Forwarding

- Deeper pipeline increases number of levels of forwarding for ALU operations
  - 4 possible sources for an ALU bypass – EX/DF, DF/DS, DS/TC, TC/WB

Floating point pipeline

- 3 functional units
  - divider, multiplier, adder
- Double precision FP ops take from 2 (negate) up to 112 cycles (square root)
- Effectively 8 stages, combined in different orders for various FP operations
  - one copy of each stage, and some instructions use a stage zero or more times, and in different orders
- Overall, rather complicated …
  - see H&P for more details
R4000 pipeline performance

• 4 major causes of pipeline stalls
  – load stalls – from using load result 1 or 2 cycles after load
  – branch stalls – 2 cycles on every taken branch, or empty branch delay slot
  – FP result stalls – RAW hazards for an FP operand
  – FP structural stalls – from conflicts for functional units in FP pipeline

SPEC92 benchmarks

Assuming a perfect cache – 5 integer and five FP programs
Dynamically scheduled pipelines

- We’ll cover this, and the scoreboard technique, in Unit 4
  - need some general background first

Pitfalls

- Unexpected hazards do occur …
  - for example, when a branch is taken before a previous instruction finishes
- Extensive pipelining can slow a machine down, or lead to worse cost-performance
  - more complex hardware can cause a longer clock cycle, killing the benefits of more pipelining
Pitfalls (cont.)

- A poor compiler can make a good machine look bad
  - compiler writers need to understand the architecture in order to
    • optimize efficiently and
    • avoid hazards
  - better to eliminate useless instructions, than make them run faster