Computer Systems Architecture
CMSC 411
Unit 3 – Instruction Pipelining

Alan Sussman
September 23, 2003

Administrivia

• HW #2 due today – solutions posted later today
• HW #1 solutions posted
• Quiz Thursday – Units 1 & 2
• Read Appendix A
• HW #1 problem 1.17d
  – MFLOPs with coprocessor
  – answer shows that MFLOPs is computed as
    (# fp ops)/(time for fp ops) = (# fp ops)/(total time – time for integer ops)
  – that is correct – don’t count integer ops against MFLOPs
  – but both are counted in MIPS (both integer and fp ops are instructions!

Last time

• Compiler/architecture interaction
  – providing a good target for the compiler can make a huge difference in performance – up to a factor of 10 on an f.p. intensive application
  – provide regularity, primitives, make costs of code sequences easy to determine
• MIPS/MIPS64 architectures
  – load/store, 64 bits (with 32-bit ops), 3 instruction formats for MIPS64 (all 32 bits), immediate and displacement addressing modes

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.
    - IR ← Mem[PC]
    - PC ← 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.)
  - A ← Regs[IR$_{6..10}$]
  - B ← Regs[IR$_{11..15}$]
  - Imm ← ((IR$_{16}$) 16##IR$_{16..31}$)

MIPS w/o pipelining (cont.)

– Execution cycle (EC): Use the ALU
  – If memory reference:
    - ALUOutput ← A + Imm
  – If register-register ALU instruction:
    - ALUOutput ← A op B
  – If register-immediate ALU instruction:
    - ALUOutput ← A op Imm
  – If branch instruction: compute the branch address and check the branch condition:
    - ALUOutput ← PC + (Imm << 2)
    - Cond ← (A op 0) (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: $LMD ← Mem[ALUOutput]$
  - Store: Mem[ALUOutput] ← B
  - Branch: if Cond then $PC ← ALUOutput$
  else $PC$ is ok

MIPS w/o pipelining (cont.)

– Write-back cycle (WB): update the registers
  - Register-register ALU instruction:
    - Regs[IR$_{16..20}$] ← ALUOutput
  - Register-immediate ALU instruction:
    - Regs[IR$_{11..15}$] ← ALUOutput
  - Load instruction:
    - Regs[IR$_{11..15}$] ← $LMD$
**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>Clock cycle</th>
<th>Inst. #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>IF</td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
</tr>
</tbody>
</table>

**Communications paths and timing**

**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.
    – Time = 50 ns + 99*10 ns = 1040 ns
  • Speedup = 4000/1040 = 3.85

Even more realistic case

• Example 3: The original MIPS implementation doesn’t always need to use the MEM cycle
  – IF -10ns
  – ID - 8ns
  – EX - 7ns
  – MEM - 10ns
  – WB - 5ns
  • 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

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>EXIDIF</th>
<th>MEMEXIDIF</th>
<th>WBMEMEXIDIF</th>
<th>WBMEMEXIF</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IF</td>
</tr>
<tr>
<td>2</td>
<td>IF</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>ID</td>
</tr>
<tr>
<td>3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>EXIDIF</td>
</tr>
<tr>
<td>4</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>MEM</td>
<td>MEMEXIDIF</td>
</tr>
<tr>
<td>5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WBMEMEXIDIF</td>
</tr>
<tr>
<td>6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td>WBMEMEXIF</td>
</tr>
</tbody>
</table>

Example 4: In Example 3, 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.
- 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 = 3300/1090 = 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. Later instruction reads it.
  - Problem: an old value may be read.
- WAW: write after write
  - One instruction writes a value. 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. 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

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

Administrivia

- HW #2 solutions posted
- Quiz today – Units 1 & 2
- Read Appendix A
- Questions about Unit 2 (or 1)?
  – including benchmark results presented by Dr. Tseng
Last time

- Pipelining
  - overlap execution of multiple instructions, to improve throughput
  - 5 execution stages in MIPS
    - IF - instruction fetch
    - ID - instruction decode, get operands
    - EX - execute
    - MEM - memory access, for loads/stores
    - WB - write back to registers
  - Pipeline costs include synchronizing stages, and having all instructions go through all stages
  - Last problem is hazards, that stall the pipeline
    - structural, data and control

No forwarding – Figure A.6

Forwarding – Fig. A.7

Forwarding (cont.)

- Compilers need to be smart enough to prevent stalls when possible
  
  Example:
  
  \[
  a = b + c + d; \\
  e = d - f; \\
  \]

  - Need to make sure that the first ADD operation delays until b and c are loaded

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

- 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
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 can fetch the correct instruction

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successor</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>+1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successor</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+2</td>
<td></td>
<td></td>
<td></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 $0.7 + 0.6 = 1.3$ cycles per instruction
  - Worse by 30%

Computer Systems Architecture
CMSC 411
Unit 3 – Instruction Pipelining

Alan Sussman
September 30, 2003

Administrivia

- Quiz 1
  - Mean: 60  Median: 59  25%: 42  75%: 83
  - questions?
- Homework #3 posted
  - due October 9

Last time

- Pipelining
  - forwarding helps reduce stalls/bubbles
  - compiler can sometimes reorder instructions to reduce stalls
    - not past branches (yet)
    - if instructions independent
  - control hazards from (usually taken) branches
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

<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. j+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst. j+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst. j+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst. j+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>idle</th>
<th>idle</th>
<th>idle</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst. j+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.L. + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>B.L. + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Compiler approaches (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

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**

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Synch. vs. asynch.</th>
<th>User request vs. coerced</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. coerced</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>Async</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Power failure</td>
<td>Async</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Terminate</td>
</tr>
</tbody>
</table>

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!

Administrivia

- Homework #3 due next Thursday
Last time

- Pipelining
  - dealing with branch delays
    - predict not taken (or taken)
    - branch delay slot – fill with instruction from before branch, from branch target, or from fall-through
    - cancelling branches – if prediction wrong
  - dealing with exceptions/faults/interrupts
    - synchronous vs. asynchronous
    - user requested vs. coerced
    - user maskable vs. non-maskable
    - within vs. between instructions
    - terminating vs. resuming

Examples of exception handling

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

Figure A.30
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.31

Functional units - Fig. A.31

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

Examples (cont.) - Fig. A.32

FP stalls from RAW hazards – Fig. A.33
**Long instructions (cont.)**

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

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

- A structural hazard

**WAW structural hazard – Fig. A.34**

<table>
<thead>
<tr>
<th>MUL.D</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>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>IF</td>
<td>ID</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F2,F4</td>
<td>IF</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>LD</td>
<td>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>
</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:

- 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.

MIPS R4000 pipeline stages

- **IF** – 1st half instruction fetch
  - PC selection and start instruction cache access
- **IS** – 2nd 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** – 1st half data fetch
  - 1st half of data cache access
- **DS** – 2nd half data fetch
  - Complete data cache access
- **TC** – tag check
  - Determine whether data cache access *hit*
- **WB** – write back for loads and ALU operations

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 (cont.)

A 2 cycle load delay – Fig. A.38

Computer Systems Architecture
CMSC 411
Unit 3 – Instruction Pipelining

Alan Sussman
October 7, 2003
Administrivia

- Homework #3 due Thursday

Last time

- Pipelining

MIPS R4000 pipeline (cont.)

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

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