Lecture Outline

- Finish ISAs (Compiler Impact)
- Basic Implementation Review
- Advantages of Pipelining
- “Easy” Challenges of Pipelining
  - Hazards (Structural, Data, Control)
- “Hard” Challenges of Pipelining
  - Maintaining Precise Exceptions
Compiler Optimizations

- High-level optimizations
  - Done on source, may be source-to-source conversions
  - Examples – map data for cache efficiency, remove conditions, etc.
- Local Optimizations
  - Optimize code in small straight-line sections
- Global Optimizations
  - Extend local opts across branches and do loop optimizations (loop unrolling)
- Register Allocation
  - Assign temporary values to registers, insert spill code
Compilers and the ISA

- Architects can help compiler writers
  - Providing regularity (already discussed)
  - Primitives, not solutions (HLL-support has not succeeded)
  - Simplify trade-offs among alternatives
  - Provide instructions that bind compile-time constants

Compiler support for MM ISAs

- Actually there is very little
- Surprising because vector-computers have good compiler support
- Problems
  - Short, architecture-limited vectors
  - Few registers and simple addressing modes
    - Vector machines support strided addressing and gather/scatters
  - Most programming languages don’t support subwords
  - Results: Only some kernels tend to be hand-coded
Implementation Review

• First, let’s think about how different instructions get executed

Instruction Fetch | Instruction Decode | Register Fetch |
------------------|-------------------|---------------|
ALU Ops           | Memory Ops        | Control Ops   |
                  | Calculate Eff. Addr |               |
                  | Memory Access     |               |
                  | Write Result      |               |
                  | Calculate Eff. Addr |               |
                  | Branch Complete   |               |
                  | Execute           |               |
                  | Write Result      |               |

All Instructions

Instruction Fetch

• Send the Program Counter (PC) to memory
• Fetch the current instruction from memory
  – IR <= Mem[PC]
• Update the PC to the next sequential
  – PC <= PC + 4 (4-bytes per instruction)
• Optimizations
  – Instruction Caches, Instruction Prefetch
• Performance Affected by
  – Code density, Instruction size variability (CISC/RISC)
Abstract Implementation

Instruction Decode/Reg Fetch

- Decide what type of instruction we have
  - ALU, Branch, Memory
  - Decode Opcode
- Get operands from Reg File
  - A <= Regs[IR_{25..21}]; B <= Regs[IR_{20..16}];
  - Imm <= SignExtend(IR_{15..0})
- Performance Affected by
  - Regularity in instruction format, instruction length
Calculate Effective Address: Memory Ops

- Calculate Memory address for data
- $\text{ALU}_{\text{output}} <= A + \text{Imm}$
- $\text{LW} \quad R10, 10(R3)$

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
</table>

Calculate Effective Address: Branch/Jump Ops

- Calculate target for branch/jump operation
- $\text{BEQZ}$, $\text{BNEZ}$, $\text{J}$
  - $\text{ALU}_{\text{output}} <= \text{NPC} + \text{Imm}; \text{cond} <= A \text{ op } 0$
  - “op” is a check against 0, equal, not-equal, etc.
  - $J$ is an unconditional
- $\text{ALU}_{\text{output}} <= A$
Execution: ALU Ops

- Perform the computation
- Register-Register
  - $ALU_{output} <= A \text{ op } B$
- Register-Immediate
  - $ALU_{output} <= A \text{ op } Imm$
- No ops need to do effective address calc and perform an operation on data
- Why?

Memory Access

- Take effective address, perform Load or Store
- Load
  - $LMD <= Mem[ALU_{output}]$
- Store
  - $Mem[ALU_{output}] <= B$
Mem Phase on Branches

- Set PC to the calculated effective address
- BEQZ, BNEZ
  - If (cond) PC <= ALU_output else PC <= NPC

Write-Back

- Send results back to register file
- Register-register ALU instructions
  - Regs[IR_{15..11}] <= ALU_output
- Register-Immediate ALU instruction
  - Regs[IR_{20..16}] <= ALU_output
- Load Instruction
  - Regs[IR_{20..16}] <= LMD
- Why does this have to be a separate step?
What is Pipelining?

- Implementation where multiple instructions are simultaneously overlapped in execution
  - Instruction processing has N different stages
  - Overlap different instructions working on different stages
- Pipelining is not new
  - Ford’s Model-T assembly line
  - Laundry – Washer/Dryer
  - IBM Stretch [1962]
  - Since the ’70s nearly all computers have been pipelined
Pipelining Advantages

- Unpipelined
  - Time
  - Latency

- Pipelined
  - Latency
  - 1/Throughput

Ideal Pipelining Performance

- Assume instruction execution takes N stages
  - Each stage takes $t_n$ time
  - Single Instruction latency, $T = \Sigma t_n$
  - Throughput = $1/T$
  - M-Instruction Latency = $M/T$

- For an N-stage pipeline
  - Single Instruction latency, $T = \Sigma t_n$
  - Throughput = $1/\max(t_n) \leq T/N$ (unless all $t_n$ are equal)
  - M-instruction Latency = $M * \max(t_n) \leq M*T/N$

- $\text{CPI}_{\text{Ideal}} = \frac{\text{CPI}_{\text{withoutpipeline}}}{\text{Pipeline Depth}}$
Why is this not the case?

- Two things we are missing
  - Pipelining overhead (latches, clock skew, jitter)
    - This eats into the maximum speedup
  - Hazards
- $\text{CPI}_{\text{Real}} = \text{CPI}_{\text{Ideal}} + \text{CPI}_{\text{Stall}}$

IBM POWER4 Clock Skew Measurement

---

How are clocks distributed?

POWER4 Clock Distribution Net

---
Recall from Earlier…

Now the pipelined version
**Pipeline Hazards**

- **Hazards**
  - Situations that prevent the next instruction from executing in its designated clock cycle
- **Structural Hazards**
  - When two different instructions want to use the same hardware resource in the same cycle (resource conflict)
- **Data Hazards**
  - When an instruction depends on the result of a previous instruction that exposes overlapping of instructions
- **Control Hazards**
  - Pipelining of PC-modifying instructions (branch, jump, etc)
How to resolve hazards?

- Simple Solution: Stall the pipeline
  - Stops some instructions from executing
  - Make them wait for older instructions to complete
  - Simple implementation to “freeze” (de-assert write-enable signals on pipeline latches)
  - Inserts a “bubble” into the pipe
  - Must propagate upstream as well! Why?

Structural Hazards

- Two cases when this can occur
  - Resource used more than once in a cycle (Memory, ALU)
  - Resource is not fully pipelined (FP Unit)
- Imagine that our pipeline shares I- and D-memory

<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>lw R10, 10(R1)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>sub R11, R2, R3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>add R12, R4, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>add R13, R6, R7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Structural Hazards
Solutions

• Stall
  – Low Cost, Simple (+)
  – Increases CPI (-)
  – Try to use for rare events in high-performance CPUs

• Duplicate Resources
  – Decreases CPI (+)
  – Increases cost (area), possibly cycle time (-)
  – Use for cheap resources, frequent cases
    • Separate I-, D-caches, Separate ALU/PC adders, Reg File Ports

Structural Hazards
Solutions

• Pipeline Resources
  – High performance (+)
  – Control is simpler than duplication (+)
  – Tough to pipeline some things (RAMs) (-)
  – Use when frequency makes it worthwhile
  – Ex. Fully pipelined FP add/multiplies critical for scientific

• Good news
  – Structural hazards don’t occur as long as each instruction uses a resource
    • At most once
    • Always in the same pipeline stage
    • For one cycle
  – RISC ISAs are designed with this in mind, reduces structural hazards
Pipeline Stalls

• What could the performance impact of unified instruction/data memory be?

Loads ~15% of instructions, Stores ~10%

Prob (Ifetch + Dfetch) = .25

\[ CPI_{\text{Real}} = CPI_{\text{Ideal}} + CPI_{\text{Stall}} = 1.0 + .25 = 1.25 \]

Data Hazards

• Two operands from different instructions use the same storage location
• Must appear as if instructions are executed to completion one at a time
• Three types of Data Hazards
  – Read-After-Write (RAW)
    • True data-dependence (Most important)
  – Write-After-Read (WAR)
  – Write-After-Write (WAW)
RAW Example

<table>
<thead>
<tr>
<th>Cycle</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>
</tr>
</thead>
<tbody>
<tr>
<td>Add R3, R2, R1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R4, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R6, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R7, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- First Add writes to R3 in cycle 5
- Second Add reads R3 in cycle 3
- Third Add reads R3 in cycle 4
  - We would compute the wrong answer because R3 holds the “old” value

Solutions to RAW Hazards

- As usual, we have a couple of choices
- Stall whenever we have a RAW
  - Huge performance penalty, dependencies are common!
- Use Bypass/Forwarding to minimize the problem
  - Data is ready by end of EXE (Add) or MEM (Load)
  - Basic idea:
    - Add comparator for each combination of destination and source registers that can have RAW hazards (How many?)
    - Add muxes to datapath to select proper value instead of regfile
  - Only stall when absolutely necessary
Solutions to RAW Hazards: Pipeline Interlocks

- Two part problem: Detect the RAW, forward/stall the pipe
  - Need to keep register ID’s along with pipe stages
  - Use comparators to check for hazards
- Operand 2 bypass ADD R1, R2, R3
  If (R3 == RD(MEM)) use ALUOUT(MEM)
  else (if R3 == RD(WB)) use ALUOUT (WB)
  else Use R3 from Register File
Forwarding, Bypassing

- Code is now “stall-free”
- Are there any cases where we must stall?

<table>
<thead>
<tr>
<th>Cycle</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>
</tr>
</thead>
<tbody>
<tr>
<td>Add R3, R2, R1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R4, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R6, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R7, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load Use Hazards

- Unfortunately, we can’t forward “backward in time”

<table>
<thead>
<tr>
<th>Cycle</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>
</tr>
</thead>
<tbody>
<tr>
<td>lw R3, 10(R1)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R4, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add R6, R3, R5</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Load Use Hazards

- Can the compiler help out?
  - Scheduling to avoid load followed by immediate use
- “Delayed Loads”
  - Define the pipeline slot after a load to be a “delay slot”
  - NO interlock hardware. Machine assumes the correct compiler
- Compiler attempts to schedule code to fill delay slots
- Limits to this approach:
  - Only can reorder between branches (5-6 instructions)
  - Order of loads/stores difficult to swap (alias problems)
  - Makes part of implementation architecturally visible

Instruction Scheduling Example

\[
a = b + c; \\
d = e - f;
\]

<table>
<thead>
<tr>
<th>No Scheduling Version</th>
<th>Scheduled Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW Rb, b</td>
<td>LW Rb, b</td>
</tr>
<tr>
<td>LW Rc, c</td>
<td>LW Rc, c</td>
</tr>
<tr>
<td>ADD Ra, Rb, Rc</td>
<td>LW Re, e</td>
</tr>
<tr>
<td>SW a, Ra</td>
<td>ADD Ra, Rb, Rc</td>
</tr>
<tr>
<td>LW Re, e</td>
<td>LW Rf, f</td>
</tr>
<tr>
<td>LW Rf, f</td>
<td>SW a, Ra</td>
</tr>
<tr>
<td>SUB Rd, Re, Rf</td>
<td>SUB Rd, Re, Rf</td>
</tr>
<tr>
<td>SW d, Rd</td>
<td>SW d, Rd</td>
</tr>
</tbody>
</table>
Other Data Hazards: WARs

- **Write-After-Read (WAR) Hazards**
  - Can’t happen in our simple 5-stage pipeline because writes always follow reads
  - Preview: Late read, early write (auto-increment)
    - \( i \) \( \text{DIV} \ (R1), --, -- \)
    - \( i+1 \) \( \text{ADD} --, R1+, -- \)
  - Preview: Out-of-Order reads (OOO-execution)

Other Data Hazards: WAWs

- **Write-After-Write (WAW) Hazards**
  - Can’t happen in our simple 5-stage pipeline because only one writeback stage (ALU ops go through MEM stage)
  - Preview: Slow operation followed by fast operation
    - \( i \) \( \text{DIVF} \ F0, --, -- \)
    - \( i+1 \) \( \text{BFPT} --, --, -- \)
    - \( i+2 \) \( \text{ADDF} \ F0, --, -- \)
  - Also cache misses (they can return at odd times)
- **What about RARs?**
Control Hazards

- In base pipeline, branch outcome not known until MEM
- Simple solution – stall until outcome is known
- Length of control hazard is branch delay
  - In this simple case, it is 3 cycles (assume 10% cond. branches)
  - $CPI_{Real} = CPI_{Ideal} + CPI_{Stall} = 1.0 + 3 \text{ cycles} \times 0.1 = 1.3$

Control Hazards: Solutions

Fast Branch Resolution

- Performance penalty could be more than 30%
  - Deeper pipelines, some code is very branch heavy
- Fast Branch Resolution
  - Adder in ID for PC + immediate targets
  - Only works for simple conditions (compare to 0)
  - Comparing two register values could be too slow
Control Hazards: Branch Characteristics

- Integer Benchmarks: 14-16% instructions are conditional branches
- FP: 3-12%
- On Average:
  - 67% of conditional branches are “taken”
  - 60% of forward branches are taken
  - 85% of backward branches are taken
  - Why?

Control Hazards: Solutions

1. Stall Pipeline
   - Simple, No backing up, No Problems with Exceptions
2. Assume not taken
   - Speculation requires back-out logic:
     - What about exceptions, auto-increment, etc
     - Bets the “wrong way”
3. Assume taken
   - Doesn’t help in simple pipeline! (don’t know target)
4. Delay Branches
   - Can help a bit… we’ll see pro’s and con’s soon
Control Hazards: Assume Not Taken

<table>
<thead>
<tr>
<th>Cycle</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>
</tr>
</thead>
<tbody>
<tr>
<td>Untaken Branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr +1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr +2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Looks good if we’re right!

<table>
<thead>
<tr>
<th>Cycle</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>
</tr>
</thead>
<tbody>
<tr>
<td>Taken Branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr +1</td>
<td>IF</td>
<td>flush</td>
<td>flush</td>
<td>flush</td>
<td>flush</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch Target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch Target +1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Control Hazards: Branch Delay Slots

- Find one instruction that will be executed no matter which way the branch goes
- Now we don’t care which way the branch goes!
  - Harder than it sounds to find instructions
- What to put in the slot (80% of the time)
  - Instruction from before the branch (indep. of branch)
  - Instruction from taken or not-taken path
    - Always safe to execute? May need clean-up code (or nullifying branches)
    - Helps if you go the right way
- Slots don’t help much with today’s machines
  - Interrupts are more difficult (why? We’ll see soon)
Now for the hard stuff!

- Precise Interrupts
  - What are interrupts?
  - Why do they have to be precise?

- Must have well-defined state at interrupt
  - All older instructions are complete
  - All younger instructions have not started
  - All interrupts are taken in program order

Interrupt Taxonomy

- Synchronous vs. Asynchronous (HW error, I/O)
- User Request (exception?) vs. Coerced
- User maskable vs. Nonmaskable (Ignorable)
- Within vs. Between Instructions
- Resume vs. Terminate

The difficult exceptions are resumable interrupts within instructions
- Save the state, correct the cause, restore the state, continue execution
Interrupt Taxonomy

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Sync vs. Async</th>
<th>User Request vs. Coerced</th>
<th>User mask vs. Nonmask</th>
<th>Within vs. BetweenInsn</th>
<th>Resume vs. terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O Device Req.</td>
<td>Async</td>
<td>Coerced</td>
<td>Nonmask</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Invoke O/S</td>
<td>Sync</td>
<td>User</td>
<td>Nonmask</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Tracing Instructions</td>
<td>Sync</td>
<td>User</td>
<td>Maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Sync</td>
<td>User</td>
<td>Maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Arithmetic Overflow</td>
<td>Sync</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Page Fault (not in main m)</td>
<td>Sync</td>
<td>Coerced</td>
<td>Nonmask</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Misaligned Memory</td>
<td>Sync</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Mem. Protection Violation</td>
<td>Sync</td>
<td>Coerced</td>
<td>Nonmask</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Using Undefined Insns</td>
<td>Sync</td>
<td>Coerced</td>
<td>Nonmask</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Hardware/Power Failure</td>
<td>Async</td>
<td>Coerced</td>
<td>Nonmask</td>
<td>Within</td>
<td>Terminate</td>
</tr>
</tbody>
</table>

Interrupts on Instruction Phases

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>IF</th>
<th>ID</th>
<th>EXE</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Overflow</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Page Fault (not in main memory)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Misaligned Memory</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Mem. Protection Violation</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

- Exceptions can occur on many different phases
- However, exceptions are only handled in WB
- Why?

load

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
</table>
| add

Computer Science 146
David Brooks
How to take an exception?

1. Force a trap instruction on the next IF
2. Squash younger instructions (Turn off all writes (register/memory) for faulting instruction and all instructions that follow it)
3. Save all processor state after trap begins
   • PC-chain, PSW, Condition Codes, trap condition
   • PC-chain is length of the branch delay plus 1
4. Perform the trap/exception code then restart where we left off

Summary of Exceptions

• Precise interrupts are a headache!
• All architected state must be precise
• Delayed branches
• Preview: Out-of-Order completion
  – What if something writes-back earlier than the exception?
• Some machines punt on the problem
  – Precise exceptions only for integer pipe
  – Special “precise mode” used for debugging (10x slower)
For next time

- Multi-cycle operations
  - More WAR, WAW nastiness
  - More precise interrupt nastiness
- SuperScalar/Dynamic Scheduling