Outline

- Announcements
  - Should be done with assignment 0 today.
  - Homework Assignment 1 out today, due back in one week: February 8th
  - Lab Assignment 1 out today, due back in two weeks: February 15th

- Pipelining

/ Hennessy/Patterson CA:AQA (3rd Edition): Appendix A

Recap

- Computers execute billions of instructions, so instruction throughput is what matters
- Main idea behind pipelining: Divide instruction execution across several stages
  - each stage accesses only a subset of the CPU’s resources
- Example: Classic 5-stage RISC pipeline
  - IF ID EX MEM WB
- Simultaneously have different instructions in different stages
  - Ideally, can issue a new instruction every cycle

Recap (Cont’d)

Pipelined Implementation of a RISC ISA

Pipeline stage: Instruction Fetch (IF)
Pipeline stage: Instruction Decode (ID)

- ID/EX.A ≠ Reg[ID/ID.IR[rs]]
- ID/EX.B ≠ Reg[ID/ID.IR[rt]]
- ID/EX.NPC ≠ ID NPC
- ID/EX.IR ≠ Reg[ID/ID.IR]
- ID/EX.Imm ≠ sign-extend (IF/ID.IR[immediate field])

Pipeline stage: Execute (EX)

- ALU instruction
  - MEM/WR.B: ALU instruction
  - MEMORY Access
  - MEM/WR.IR ≠ EX/MEM.IR;
  - MEM/WR.ALUOutput ≠ EX/MEM.ALUoutput;
  - MEM/WR.LMD ≠ Mem[EX/MEM.ALUOutput] or Mem[EX/MEM.ALUOutput] ≠ EX/MEM.B;
- Load/store instruction
  - MEM/WR.B: ALU instruction
  - MEM/WR.IR ≠ EX/MEM.IR;
  - MEM/WR.ALUOutput ≠ EX/MEM.ALUoutput;
  - MEM/WR.LMD ≠ Mem[EX/MEM.ALUOutput] or Mem[EX/MEM.ALUOutput] ≠ EX/MEM.B;
- Branch instruction
  - EX/MEM.ALUOutput ≠ ID/NPC + (ID/EX.Imm <<2);
  - EX/MEM.cond ≠ (ID/EX.A ==0);

Three classes of hazards:
- Structural: Hardware cannot support this combination of instructions - two instructions need the same resource.
- Data: Instruction depends on result of prior instruction still in the pipeline
- Control: Pipelining of branches & other instructions that change the PC

Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline
- To do this, hardware or software must detect that a hazard has occurred

Pipeline Hazards

- Should we expect a CPI of 1 in practice?
- Unfortunately, the answer to the question is NO.
- Limit to pipelining: Hazards
  - Prevent next instruction from executing during its designated clock cycle

Three classes of hazards:
- Structural: Hardware cannot support this combination of instructions - two instructions need the same resource.
- Data: Instruction depends on result of prior instruction still in the pipeline
- Control: Pipelining of branches & other instructions that change the PC

Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline
- To do this, hardware or software must detect that a hazard has occurred

Pipeline Hazards (A): Structural Hazards

- Occur when two or more instructions need the same resource
- Common methods for eliminating structural hazards are:
  - Duplicate resources
  - Pipe the resource
  - Reorder the instructions

- It may be too expensive to eliminate a structural hazard, in which case the pipeline should stall
  - no new instructions are issued until the hazard has been resolved

- What are some examples of structural hazards?
One Memory Port Structural Hazard

Pipeline Speedup Example: One or Two Memory Ports

• Two machines
  – Machine A: Dual ported memory
  – Machine B: Single ported memory, but its pipelined implementation has a clock rate that is 1.05 times faster
  – Ideal CPI = 1 for both
  – Loads are 40% of instructions executed (cause stalls in machine B)
• Which is faster?

\[
\text{Speedup}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times 1 = \text{Pipeline Depth}
\]

\[
\text{Speedup}_B = \frac{\text{Pipeline Depth}}{1 + 0.4 \times 1} \times 1.05 = 0.75 \times \text{Pipeline Depth}
\]

Machine A is 1.33 times faster

Data Hazards and Pipeline Stalls

• Do all kinds of data hazards translate into pipeline stalls?
  • NO, whether or not a data hazard results in a stall depends on the pipeline structure
  • For the simple five-stage RISC pipeline
    – Only RAW hazards result in a pipeline stall
      • Instruction reading a register needs to wait until it is written
    – WAR and WAW hazards cannot occur because
      • All instructions take 5 stages
      • Reads happen in the 2nd stage (ID)
      • Writes happen in the 5th stage (WB)
      • No way for a write from a subsequent instruction to interfere with the read (or write) of a prior instruction
  • For more complicated pipelines (later in the course)
    – Both WAR and WAW hazards are possible if instructions execute out of order or access (read) data later in the pipeline
Reducing Impact of RAW Hazards: Data Forwarding

- Data forwarding (also called bypassing or short-circuiting)
  - Directly transfers data from each stage to earlier pipeline stages
  - Result is accessible before it gets written into the register file

Instr i: \( \text{add } r1, r2, r3 \) (result ready after EX stage)

Instr j: \( \text{sub } r4, r1, r5 \) (result needed in EX stage)

- To support data forwarding, additional hardware is required.
  - Multiplexers to allow data to be transferred back
  - Control logic for the multiplexers

Avoidance of RAW Hazards Using Forwarding

- Data forwarding eliminates RAW hazards.

Forwarding Does Not Eliminate All Hazards

- However, forwarding does not eliminate all hazards.
  - Cope with this by stalling the EXE stage till results are available

Pipeline Hazards (C): Control Hazards

- Control hazards occur due to instructions changing the PC
  - Can result in a large performance loss

- A branch is either
  - Taken: \( \text{PC} \leftarrow \text{PC} + \text{Imm} \)
  - Not Taken: \( \text{PC} \leftarrow \text{PC} + 4 \)

- Cannot fetch the next instruction till value of PC is known

- Simplest solution is to stall the pipeline upon detecting a branch
  - ID stage detects the branch
  - Don’t know if the branch is taken until the EX stage
  - New PC is not changed until the end of the MEM stage, after determining if the branch is taken and the new PC value
  - If the branch is taken, we need to repeat some stages and fetch new instructions

(Review) Pipelined Implementation of a RISC ISA
3 Cycle Stall on Branch-Induced Control Hazards

```
beq r1, r3, 36
and r2, r3, r5
or  r6, r1, r7
add r8, r1, r9
xor r10, r1, r11
```

Impact of Branch Stalls
- If CPI = 1, 30% branches
  - Stall 3 cycles => new CPI = (1 + 0.3*3) = 1.9
  - 50% of these branches taken => new CPI = 1 + 0.15*3 + 0.15*2 = 1.7
- Penalty would be worse for current-day (longer) pipelines
  - IF and ID-like stages are each multiple-cycle
- How do we reduce impact of branch stalls?
  - Two part solution:
    - Determine branch taken or not sooner, AND
    - Compute taken branch address earlier

Branch Behavior in Programs
- Based on SPEC benchmarks on DLX (CA-AQA, 2nd Edition)
  - Branches occur with a frequency of 14% to 16% in integer programs and 3% to 12% in floating point programs.
  - About 75% of the branches are forward branches
  - 60% of forward branches are taken
  - 80% of backward branches are taken
- Why are branches (especially backward branches) more likely to be taken than not taken?

Dealing with Branch Stalls
- Approach 1: Stall until branch direction is clear
- Approach 2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - PC+4 already calculated, so use it to get next instruction; chances are the branch is not taken
  - “Squash” instructions in pipeline if branch actually taken
    - Can do this because CPV state not updated till late in the pipeline
- Approach 3: Predict Branch Taken
  - Most branches are taken
  - But haven’t yet calculated target address in a 5-stage RISC pipeline
    - No, will still incur a 1-cycle latency
    - Makes sense on machines where branch target is known before outcome
      - (later: Branch Target Buffers)
- Approach 4: Delayed Branch
  - Define branch to take place AFTER n following instructions
    - branch instruction
    - sequential successor, sequential successor, ............
    - sequential successor, branch target if taken
    - n branch delay slots

Pipelined Implementation of a RISC ISA: Reducing Branch Penalty to 1 cycle

Dealing with Branch Stalls (cont’d)
Branch Delay Slots

- Instructions in the branch delay slot(s) get executed whether or not branch is taken
- Heavily used in early RISC machines
  - 1 delay-slot suffices for a 5-stage pipeline (target available at end of ID)
  - Machines with deep pipelines require additional delay slots to avoid branch penalties
  - Benefits are unclear

Evaluating Branch Alternatives

<table>
<thead>
<tr>
<th>Scheduling scheme</th>
<th>Branch CPI</th>
<th>speedup vs. unpipelined</th>
<th>speedup vs. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow stall pipeline</td>
<td>3</td>
<td>1.42</td>
<td>3.5</td>
</tr>
<tr>
<td>Fast stall pipeline</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>0.7</td>
<td>1.10</td>
<td>4.3</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.07</td>
<td>4.7</td>
</tr>
</tbody>
</table>

- A compiler can reorder instructions to further improve speedup

Scheduling the Branch Delay Slot

Where does the instruction for the delay slot come from?

- Nullifying or cancelling branches
  - Converts delay slot instruction into a nop

Importance of Avoiding Branch Stalls

- Crucial in modern microprocessors, which issue/execute multiple instructions every cycle
  - Need to have a steady stream of instructions to keep the hardware busy
  - Stalls due to control hazards dominate
- So far, we have looked at static schemes for reducing branch penalties
  - Same scheme applies to every branch instruction
- Potential for increased benefits from dynamic schemes
  - Can choose most appropriate scheme separately for each instruction
    - Branches to top of loop have different behavior (Taken) than
    - "if (x == 0) return;" (Not Taken)
    - Can "learn" appropriate scheme based on observed behavior
  - Dynamic (hardware) branch prediction schemes
    - For both direction (T or NT) and target prediction
    - Key element of all modern microprocessors