Lecture 3
Pipelining

September 19, 2007
Outline

• Announcements
  – Homework Assignment 1 due today
  – Mini Quiz next week: Sep. 26th
    To make sure we are on the same page with the basics; Covers the following:
    • Chapter 1
    • Appendix A
    • Appendix B

• Pipelining

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
  
<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
</table>

- Simultaneously have different instructions in different stages
  - Ideally, can issue a new instruction every cycle
Recap (Cont’d)

Time (clock cycles)

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

Instr. order

i

i+1

i+2
Pipelined Implementation of a RISC ISA
Pipeline stage: Instruction Fetch (IF)

IF/ID.IR ← Mem[PC];
IF/ID.NPC,PC ←
   (if ((EX/MEM.opcode == branch) &
       EX/MEM.cond)
   {EX/MEM.ALUOutput} else {PC+4}
);

Branch target address (EX/MEM.ALUOutput register)
Branch comparison result (EX/MEM.cond register)
Pipeline stage: Instruction Decode (ID)

ID/EX.A ← Regs[IF/ID.IR[rs]];
ID/EX.B ← Regs[IF/ID.IR[rt]];
ID/EX.NPC ← IF/ID.NPC;
ID/EX.IR ← IF/ID.IR;
ID/EX.Imm ← sign-extend (IF/ID.IR[immediate field])
Pipeline stage: Execute (EX)

**ALU instruction**
- EX/MEM.IR ← ID/EX.IR;
- EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B;
  or
- EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm;

**Branch instruction**
- EX/MEM.ALUOutput ← ID/EX.NPC +
  (ID/EX.Imm <<2);
- EX/MEM.cond ←
  (ID/EX.A ==0);

**Load/store instruction**
- EX/MEM.IR ← ID/EX.IR;
- EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm;
- EX/MEM.B ← ID/EX.B;
Pipeline stage: Memory access (MEM)

**ALU instruction**
- MEM/WB.IR ← EX/MEM.IR;
- MEM.WB.ALUOutput ← EX/MEM.ALUOutput;

**Load/store instruction**
- MEM/WB.IR ← EX/MEM.IR;
- MEM/WB.LMD ← Mem[EX/MEM.ALUOutput];
  - or
- Mem[EX/MEM.ALUOutput] ← EX/MEM.B;
Pipeline stage: Write Back (WB)

ALU instruction
Regs[MEM/WB.IR[rd]] ← MEM.WB(ALUOutput);
or
Regs[MEM/WB.IR[rt]] ← MEM.WB(ALUOutput);

Load instruction only
Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD
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
  – Pipeline 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

Time (clock cycles)

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Reg</td>
<td>Ifetch</td>
</tr>
<tr>
<td>i</td>
<td>(LD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i+3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i+4</td>
</tr>
</tbody>
</table>

Instr. order

- i (LD)
- i+1
- i+2
- stall
- i+3
- i+4
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?

\[
\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]

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

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

Machine A is 1.33 times faster
Pipeline Hazards (B): Data Hazards

Three generic types of data hazards

- **Read After Write (RAW)**
  - Instr\(_j\) tries to read operand before Instr\(_i\) (I < J) writes it
  - Called a dependence

- **Write After Read (WAR)**
  - Instr\(_j\) writes operand before Instr\(_i\) reads it
  - Called an anti-dependence
    - Name dependence (renaming)
    - No value being transmitted

- **Write After Write (WAW)**
  - Instr\(_j\) writes operand before Instr\(_i\) writes it
  - Called an output dependence
    - Name dependence (renaming)
    - No value being transmitted

\[ I: \text{add } r1,r2,r3 \]
\[ J: \text{sub } r4,r1,r3 \]
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
RAW Hazards in the 5-stage Pipeline

- **add** \( r_1, r_2, r_3 \)
- **sub** \( r_4, r_1, r_3 \)
- **and** \( r_6, r_1, r_7 \)
- **or** \( r_8, r_1, r_9 \)
- **xor** \( r_{10}, r_1, r_{11} \)

*Forwarding through the register file*
Absence of WAR and WAW Hazards

**Instr. order**

- **add** r4, r1, r3  
  (WAR)
- **sub** r1, r2, r3
- **or** r8, r2, r3  
  (WAW)
- **xor** r8, r4, r5
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: \texttt{add r1, r2, r3}  \hspace{1cm} \text{ (result ready after EX stage) }

Instr j: \texttt{sub r4, r1, r5}  \hspace{1cm} \text{ (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
Hardware Changes for Forwarding
Avoidance of RAW Hazards Using Forwarding

- add \( r_1, r_2, r_3 \)
- sub \( r_4, r_1, r_3 \)
- and \( r_6, r_1, r_7 \)
- or \( r_8, r_1, r_9 \)
- xor \( r_{10}, r_1, r_{11} \)

Split-phase access
Forwarding Does Not Eliminate All Hazards

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw r1, 0(r2)</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
</tr>
<tr>
<td>sub r4, r1, r6</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
</tr>
<tr>
<td>and r6, r1, r7</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
</tr>
<tr>
<td>or r8, r1, r9</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
</tr>
</tbody>
</table>

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: PC ← PC + Imm
  - Not Taken: PC ← 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

Instr. order

beq r1,r3,36

and r2,r3,r5

or r6,r1,r7

add r8,r1,r9

xor r10,r1,r11

New target available

Branch direction known

Time (clock cycles)
Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7

Ifetch Reg ALU DMem Reg

Ifetch Reg ALU DMem Reg

Ifetch Reg ALU DMem Reg

Ifetch Reg ALU DMem Reg

Ifetch Reg ALU DMem Reg

Ifetch Reg ALU DMem Reg

Ifetch Reg ALU DMem Reg
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
Pipelined Implementation of a RISC ISA: Reducing Branch Penalty to 1 cycle
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 CPU state not updated till late in the pipeline

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Clock Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1  2  3  4  5  6  7  8  9</td>
</tr>
<tr>
<td>i (T)</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>i+1</td>
<td>IF  idle  idle  idle  idle</td>
</tr>
<tr>
<td>T</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>T+1</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>T+2</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
</tbody>
</table>
Dealing with Branch Stalls (cont’d)

- **Approach 3: Predict Branch Taken**
  - Most branches are taken
  - But haven’t yet calculated target address in a 5-stage RISC pipeline
    - So, 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_1
    sequential successor_2
    ........
    sequential successor_n
    branch target if taken
    ```
    
    n branch delay slots
Branch Delay Slots

- Instructions in the branch delay slot(s) get executed whether or not branch is taken

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Clock Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>i (T)</td>
<td>IF</td>
</tr>
<tr>
<td>D(i+1)</td>
<td>IF</td>
</tr>
<tr>
<td>T</td>
<td>IF</td>
</tr>
<tr>
<td>T+1</td>
<td>IF</td>
</tr>
<tr>
<td>T+2</td>
<td>IF</td>
</tr>
</tbody>
</table>

- 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
Scheduling the Branch Delay Slot

Where does the instruction for the delay slot come from?

(a) From before

DADD R1, R2, R3
if R2 = 0 then

Delay slot

becomes

if R2 = 0 then

DADD R1, R2, R3

(b) From target

DSUB R4, R5, R6
DADD R1, R2, R3
if R1 = 0 then

Delay slot

becomes

DSUB R4, R5, R6

(c) From fall-through

DADD R1, R2, R3
if R1 = 0 then

Delay slot

OR R7, R8, R9
DSUB R4, R5, R6

Nullifying or cancelling branches
– Converts delay slot instruction into a nop
Evaluating Branch Alternatives

**Assumptions**
- 14% of instructions are branches
- 30% of branches are not taken
- 50% of delay slots can be filled with useful instructions

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

<table>
<thead>
<tr>
<th>Scheduling scheme</th>
<th>Branch penalty</th>
<th>CPI</th>
<th>speedup v. unpipelined</th>
<th>speedup v. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow stall pipeline</td>
<td>3</td>
<td>1.42</td>
<td>3.5</td>
<td>1.0</td>
</tr>
<tr>
<td>Fast stall pipeline</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.26</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.26</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>0.7</td>
<td>1.10</td>
<td>4.5</td>
<td>1.29</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.07</td>
<td>4.7</td>
<td>1.34</td>
</tr>
</tbody>
</table>

- A compiler can reorder instructions to further improve speedup
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