Topics

- Program 3
- Pipelining (Mic-3)
  - Tanenbaum 4.4.4
- Branch prediction
  - Tanenbaum 4.5.2

Program 3: Adding IJVM Instructions

- Modify the microprogram file to implement \texttt{ISHR, ISHL, IMUL}
  - The IMUL algorithm is “challenging”
- This is a TEAM assignment. There are no “individual” portions
  - Although there are 3 instructions, the level of difficulty is not balanced
  - You should be familiar with your team members by now to partition workload and contribute fairly
- \textbf{Don’t wait until the last minute!!!}
  - I will be traveling Dec. 3 - 5
Datapath of Mic-3 Microarchitecture

- Add latches on A bus, B bus, and C bus

- Benefits
  - Speed up the clock cycle because maximum delay is shorter
  - Use all parts of datapath during every cycle

Pipelining is Natural!

Laundry Example

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold

- Washer takes 30 minutes

- Dryer takes 30 minutes

- “Folder” takes 30 minutes

- “Stasher” takes 30 minutes to put clothes into drawers

Adapted from David Patterson’s CS 252 lecture notes. Copyright 2001 UCB.
Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Adapted from David Patterson’s CS 252 lecture notes. Copyright 2001 UCB.

Pipelined Laundry: Start work ASAP

- Pipelined laundry takes 3.5 hours for 4 loads!

Adapted from David Patterson’s CS 252 lecture notes. Copyright 2001 UCB.
Pipelining Lessons

- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Use different resources to execute multiple tasks simultaneously
- Potential speedup = Number pipe stages
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

Classic Five-Stage Pipeline

- **Ifetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Perform calculation for instruction
- **Mem**: Access the data from the Data Memory
- **Wr**: Write the data back to the register file
Pipelining

- Improve performance by increasing instruction throughput
- Ideal speedup is number of stages in the pipeline
- Do we achieve this?

Conventional Pipelined Execution Representation

- Assume each stage is one cycle
- # of cycles = # of instructions + (# of stages – 1)
  - Assuming full utilization of pipeline

Adapted from David Patterson’s CS 252 lecture notes. Copyright 2001 UCB.
Example

• Assume we had 96 instructions
  – Without pipeline: $96 \times 5 = 480$ cycles
  – With pipeline: $96 + (5 - 1) = 100$ cycles

\[
\text{Speedup} = \frac{\text{Execution time before improvement}}{\text{Execution time after improvement}}
\]

\[
\text{Speedup} = \frac{480 \text{ cycles}}{100 \text{ cycles}} = 4.8
\]

• We can approach the ideal speedup
  – However, this is not realistic program flow
  – Programs contain data hazards and control hazards

Data Hazards

• Created whenever there is a dependence between instructions where pipelining overlap changes the order of access

• True dependence: Read After Write (RAW)
  – Instructions are dependent upon previous data results that aren’t available
  – Causes stalls in the pipeline to wait for data

• Other hazard types include WAW and WAR
  – Can be handled with register renaming
  – Note that RAR is not a hazard
Control Hazards

• The IFU pipelines with sequential instruction stream in method area

• A branch may indicate another control flow
  
  ILOAD i  
  BIPUSH 3  
  IF_ICMPEQ L1  
  ILOAD j  
  BIPUSH 1  
  ISUB  
  ISTORE j  
  ISTORE j  
  GOTO L2  
  L1:  BIPUSH 0  
  L2: …

The pipeline issues the next sequential instruction before resolving if the branch is taken or not

Branches

• Unconditional
  – Must decide where to fetch before knowing what instruction it just got
  – Can insert a “delay slot” after unconditional branch
    • Try to fill it with useful work, but typically a NOP

• Conditional
  – Must test the condition before knowing what instruction to fetch
  – Can stall the pipeline until the branch is resolved
    • Defeats the purpose of pipelining!
Branch Prediction

• A simple method
  – Assume backward branching is taken
  – Assume forward branching is not taken

• If prediction is correct, then the pipeline is OK

• If prediction is incorrect, then must have a way to roll back the CPU state to before mispredicted branch
  – Costly in H/W

Dynamic Branch Prediction

• **Branch History Table (BHT)**
  – CPU maintains a history of branch locations and their previous behavior

• **Branch Target Address Cache (BTAC)**
  – Stores the target location for branches predicted taken

• **Use 1 bit to store history**
  – Stores what the branch did the last time
  – With loops, you will always mispredict entering and exiting

• **Use 2 bits to store history**
  – Maintains a strong and weak prediction state
  – Must mispredict twice to change the prediction
2-Bit Branch Prediction FSM

Strongly predicts taken

Weakly predicts taken

Weakly predicts not taken

Strongly predicts not taken

Static Branch Prediction

- Compiler uses branch instructions that specify the branch’s normal outcome
  - Based upon the code to be compiled
  - Taken or Not Taken

- Program is simulated to profile the branch behavior
  - Branch statistics are given to the compiler