Pipelining Review
- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
- Data hazards must be handled carefully
- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)

Outline
- Pipelined Control
- Control Hazards
- RAW, WAR, WAW
- Brainstorm on pipeline bugs
Control Hazards

- When the flow of instruction addresses is not what the pipeline expects; incurred by change of flow instructions
  - Conditional branches (beq, bne)
  - Unconditional branches (j)

- Possible solutions
  - Stall
  - Move decision point earlier in the pipeline
  - Predict
  - Delay decision (requires compiler support)

Datapath Branch and Jump Hardware

Jumps Incur One Stall

- Jumps not decoded until ID, so one stall is needed

- Fortunately, jumps are very infrequent – only 2% of the SPECint instruction mix

Administrivia

- Finish Lab 3; meet with TA Friday
- Midterm Tue Oct 12 5:30 - 8:30 in 101 Morgan
  - Northwest corner of campus, near Arch and Hearst
  - Midterm review Sunday Oct 10, 7 PM, 306 Soda
  - Bring 1 page, handwritten notes, both sides
  - Nothing electronic: no calculators, cell phones, pagers, …
  - Meet at LaVal’s Northside afterwards for Pizza

Review: Branches Incur Three Stalls

Can fix branch hazard by waiting – stall; but affects throughput.
Moving Branch Decisions Earlier in Pipe

- Move the branch decision hardware back to the EX stage
  - Reduces the number of stall cycles to two
  - Adds an and gate and a 2x1 mux to the EX timing path
- Add hardware to compute the branch target address and evaluate the branch decision to the ID stage
  - Reduces the number of stall cycles to one (like with jumps)
  - Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)
  - Comparing the registers can’t be done until after RegFile read, so comparing and updating the PC adds a comparator, an and gate, and a 3x1 mux to the ID timing path
  - Need forwarding hardware in ID stage
- For longer pipelines, decision points are later in the pipeline, incurring more stalls, so we need a better solution

Early Branch Forwarding Issues

- Bypass of source operands from the EX/MEM

```python
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == IF/ID.RegisterRs))
    ForwardC = 1
```

```python
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == IF/ID.RegisterRt))
    ForwardD = 1
```

- MEM/WB dependency also needs to be forwarded
- If the instruction 2 before the branch is a load, then a stall will be required since the MEM stage memory access is occurring at the same time as the ID stage branch compare operation

Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome
  1. Predict not taken – always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch is taken does the pipeline stall
     - If taken, flush instructions in the pipeline after the branch
     - in IR, ID, and EX branch logic in MEM – three stalls
     - in IR if branch logic in ID – one stall
   - ensure that those flushed instructions haven’t changed machine state– automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite or RegWrite)
   - restart the pipeline at the branch destination
  2. Predict taken – always predict branches will be taken
   - Predict taken always incurs a stall (if branch destination hardware has been moved to the ID stage)
   - As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance
  3. Dynamic branch prediction – predict branches at run-time using run-time information

Flush with Misprediction (Not Taken)

```
4 beq $1,$2,2
8 sub $4,$1,$5
```

- To flush the IF stage instruction, add a IF.Flush control line that zeros the instruction field of the IF/ID pipeline register (transforming it into a `noop`)

Flush with Misprediction (Not Taken)

```
4 beq $1,$2,2
8 sub $4,$1,$5
16 and $6,$1,$7
20 or r8,$1,$9
```

- To flush the IF stage instruction, add a IF.Flush control line that zeros the instruction field of the IF/ID pipeline register (transforming it into a `noop`)

Branch Prediction, cont'

- Resolve branch hazards by statically assuming a given outcome and proceeding
  2. Predict taken – always predict branches will be taken
     - Predict taken always incurs a stall (if branch destination hardware has been moved to the ID stage)
     - As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance
  3. Dynamic branch prediction – predict branches at run-time using run-time information
Dynamic Branch Prediction

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage, addressed by the lower bits of the PC, contains a bit that tells whether the branch was taken the last time it was executed.
  - Bit may predict incorrectly (may be from a different branch with the same low order PC bits, or may be a wrong prediction for this branch) but the doesn’t affect correctness, just performance.
  - If the prediction is wrong, flush the incorrect instructions in pipeline, restart the pipeline with the right instructions, and invert the prediction bit.
- The BHT predicts when a branch is taken, but does not tell where its taken to!
  - A branch target buffer (BTB) in the IF stage can cache the branch target address (or even the branch target instruction) so that a stall can be avoided.

1-bit Prediction Accuracy

- 1-bit predictor in loop is incorrect twice when not taken
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code
  1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1).
  2. As long as branch is taken (looping), prediction is correct
  3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0).
- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time.

2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.

Delayed Decision

- First, move the branch decision hardware and target address calculation to the ID pipeline stage.
- A delayed branch always executes the next sequential instruction – the branch takes effect after that next instruction.
  - MIPS software moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay.
- As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot.
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches.
  - Growth in available transistors has made dynamic approaches relatively cheaper.

Scheduling Branch Delay Slots

- A is the best choice, fills delay slot & reduces instruction count (IC).
- In B, the sub instruction may need to be copied, increasing IC.
- In B and C, must be okay to execute sub when branch fails.
3 Generic Data Hazards: RAW, WAR, WAW

• **Read After Write (RAW)**
  Instr\textsubscript{J} tries to read operand before Instr\textsubscript{I} writes it
  
  \begin{align*}
  \text{I: } & \text{add } r1, r2, r3 \\
  \text{J: } & \text{sub } r4, r1, r3
  \end{align*}

  • Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.
  • Forwarding handles many, but not all, RAW dependencies in 5 stage MIPS pipeline

• **Write After Read (WAR)**
  Instr\textsubscript{J} writes operand before Instr\textsubscript{I} reads it
  
  \begin{align*}
  \text{I: } & \text{sub } r4, r1, r3 \\
  \text{J: } & \text{add } r1, r2, r3 \\
  \text{K: } & \text{mul } r6, r1, r7
  \end{align*}

  • Called an “anti-dependence” by compiler writers. This results from “reuse” of the name “r1”.
  • Can’t happen in MIPS 5 stage pipeline because:
    – All instructions take 5 stages, and
    – Reads are always in stage 2, and
    – Register Writes must be in stage 5

3 Generic Data Hazards: RAW, WAR, WAW

• **Write After Write (WAW)**
  Instr\textsubscript{J} writes operand before Instr\textsubscript{I} writes it.
  
  \begin{align*}
  \text{I: } & \text{sub } r1, r4, r3 \\
  \text{J: } & \text{add } r1, r2, r3 \\
  \text{K: } & \text{mul } r6, r1, r7
  \end{align*}

  • Called an “output dependence” by compiler writers. This also results from the “reuse” of name “r1”.
  • Can’t happen in MIPS 5 stage pipeline because:
    – All instructions take 5 stages, and
    – Register Writes must be in stage 5
  • Can see WAR and WAW in more complicated pipes

**Brain storm on pipeline bugs**

• Where are bugs likely to hide in a pipelined processor?
  1. ...
  2. ...

• How can you write tests to uncover these likely bugs?
  1. ...
  2. ...

• Once it passes a test, never need to run it again in the design process?
• Suppose we use a 4 stage pipeline that combines memory access and write back stages for all instructions but load, stalling when there are structural hazards. Impact?

1. The branch delay slot is now 0 instructions
2. Most loads cause stall since often a structural hazard on reg. writes
3. Most stores cause stall since they have a structural hazard
4. Both 2 & 3: most loads/stores cause stall due to structural hazards
5. Most loads cause stall, but there is no load-use hazard anymore
6. Both 2 & 3, but there is no load-use hazard anymore
7. None of the above

Q: Why not say every load stalls?
A: Not all next instructions write in Wr stage

Summary: Designing a Pipelined Processor
• Go back and examine your data path and control diagram
• Associate resources with states
  – Be sure there are no structural hazards: one use / clock cycle
• Add pipeline registers between stages to balance clock cycle
  – Amdahl's Law suggests splitting longest stage
• Resolve all data and control dependencies
  – If backwards in time in pipeline drawing to registers
    => data hazard: forward or stall to resolve them
  – If backwards in time in pipeline drawing to PC
    => control hazard: we’ll see next time
• 5 stage pipeline with reads early in same stage, writes later in same stage, avoids WAR/WAW hazards
• Assert control in appropriate stage
• Develop test instruction sequences likely to uncover pipeline bugs (If you don’t test it, it won’t work)