Introduction to Pipelining: Datapath and Control

March 17, 2003
John Kubiatowicz (www.cs.berkeley.edu/~kubitron)

lecture slides: http://inst.eecs.berkeley.edu/~cs152/

Recap: Sequential Laundry

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Recap: Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences

Recap: Ideal Pipelining

Assume instructions are completely independent!

<table>
<thead>
<tr>
<th>Task Order</th>
<th>IF</th>
<th>DCD</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>IF</td>
<td>DCD</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>B</td>
<td>IF</td>
<td>DCD</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>C</td>
<td>IF</td>
<td>DCD</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>D</td>
<td>IF</td>
<td>DCD</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Maximum Speedup ≤ Number of stages

Speedup ≤ Time for unpipelined operation
Time for longest stage

Example: 40ns data path, 5 stages, Longest stage is 10 ns, Speedup ≤ 4
The Big Picture: Where are We Now?

° The Five Classic Components of a Computer

Today’s Topics:
- Recap last lecture/finish datapath
- Pipelined Control/ Do it yourself Pipelined Control
- Administrivia
- Hazards/Forwarding
- Exceptions
- Review MIPS R3000 pipeline

Recap: Can pipelining get us into trouble?
° Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
  - data hazards: attempt to use item before it is ready
    - E.g., one sock of pair in dryer and one in washer; can’t fold until get sock from washer through dryer
    - instruction depends on result of prior instruction still in the pipeline
  - control hazards: attempt to make a decision before condition is evaluated
    - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in branch instructions

° Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards

Single Memory is a Structural Hazard

Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards limit performance
° Example: if 1.3 memory accesses per instruction and only one memory access per cycle then
  - average CPI ≥ 1.3
  - otherwise resource is more than 100% utilized
Control Hazard Solution #1: Stall

- **Stall**: wait until decision is clear
- **Impact**: 2 lost cycles (i.e. 3 clock cycles per branch instruction) => slow
- **Move decision to end of decode**
  - save 1 cycle per branch

Control Hazard Solution #2: Predict

- **Predict**: guess one direction then back up if wrong
- **Impact**: 0 lost cycles per branch instruction if right, 1 if wrong (right - 50% of time)
  - Need to “Squash” and restart following instruction if wrong
  - Produce CPI on branch of (1 * .5 + 2 * .5) = 1.5
  - Total CPI might then be: 1.5 * .2 + 1 * .8 = 1.1 (20% branch)
- **More dynamic scheme**: history of 1 branch (- 90%)

Control Hazard Solution #3: Delayed Branch

- **Delayed Branch**: Redefine branch behavior (takes place after next instruction)
- **Impact**: 0 clock cycles per branch instruction if can find instruction to put in “slot” (- 50% of time)
- **As launch more instruction per clock cycle, less useful**

Data Hazard on r1: Read after write hazard (RAW)

- `add r1,r2,r3`
- `sub r4,r1,r3`
- `and r6,r1,r7`
- `or r8,r1,r9`
- `xor r10,r1,r11`
Data Hazard on r1: Read after write hazard (RAW)

- Dependencies backwards in time are hazards

```
<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
</tr>
<tr>
<td>sub r4, r1, r3</td>
</tr>
<tr>
<td>and r6, r1, r7</td>
</tr>
<tr>
<td>or r8, r1, r9</td>
</tr>
<tr>
<td>xor r10, r1, r11</td>
</tr>
</tbody>
</table>
```

Data Hazard Solution: Forwarding

- “Forward” result from one stage to another

```
<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
</tr>
<tr>
<td>sub r4, r1, r3</td>
</tr>
<tr>
<td>and r6, r1, r7</td>
</tr>
<tr>
<td>or r8, r1, r9</td>
</tr>
<tr>
<td>xor r10, r1, r11</td>
</tr>
</tbody>
</table>
```

Forwarding (or Bypassing): What about Loads?

- Dependencies backwards in time are hazards

```
<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw r1, 0(r2)</td>
</tr>
<tr>
<td>sub r4, r1, r3</td>
</tr>
</tbody>
</table>
```

- Can’t solve with forwarding:
- Must delay/stall instruction dependent on loads

Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards

```
<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw r1, 0(r2)</td>
</tr>
<tr>
<td>sub r4, r1, r3</td>
</tr>
</tbody>
</table>
```

- Can’t solve with forwarding:
- Must delay/stall instruction dependent on loads
Recap: Data Hazards

<table>
<thead>
<tr>
<th>I-Fetch</th>
<th>DCD</th>
<th>MemOpFetch</th>
<th>OpFetch</th>
<th>Exec</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFetch</td>
<td>DCD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Structural Hazard

<table>
<thead>
<tr>
<th>I-Fetch</th>
<th>DCD</th>
<th>OpFetch</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFetch</td>
<td>DCD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Control Hazard

WAR Data Hazard (write after read)

<table>
<thead>
<tr>
<th>IF</th>
<th>DCD</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>DCD</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>

RAW (read after write) Data Hazard

<table>
<thead>
<tr>
<th>IF</th>
<th>DCD</th>
<th>EX</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>DCD</td>
<td>Mem</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

Designing a Pipelined Processor

- Go back and examine your datapath and control diagram
- associated resources with states
- ensure that flows do not conflict, or figure out how to resolve conflicts
- assert control in appropriate stage

Control and Datapath: Split state diag into 5 pieces

Control and Datapath: Split state diag into 5 pieces

Pipelined Processor (almost) for slides

- What happens if we start a new instruction every cycle?
Pipelined Datapath (as in book): hard to read

### Administrivia

- **Midterm Results:**
  - Avg: 70, StdDev = 9.5, Max = 88, Low = 50
- **Lab 4: How are you doing?**
  - Control: Behavioral Verilog. Probably using CASE statement
  - Data: Schematics on top of your components
  - High-level simulation!

### Pipelining the Load Instruction

- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File’s Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File’s Write port (bus W) for the Wr stage

### The Four Stages of R-type

- **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec:**
  - ALU operates on the two register operands
  - Update PC
- **Wr:** Write the ALU output back to the register file
**Pipelining the R-type and Load Instruction**

- **Clock:** Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 | Cycle 8 | Cycle 9

  - **R-type:** Ifetch | Reg/Dec | Exec | Wr
  - **Load:** Ifetch | Reg/Dec | Exec | Mem | Wr
  - **R-type:** Ifetch | Reg/Dec | Exec | Wr

- **Important Observation**
  - **Each functional unit can only be used once per instruction.**
  - **Each functional unit must be used at the same stage for all instructions:**
    - Load uses Register File’s Write Port during its 5th stage
    - R-type uses Register File’s Write Port during its 4th stage

- **Ops! We have a problem!**

- We have pipeline conflict or structural hazard:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

- **Solution 1: Insert “Bubble” into the Pipeline**
  - **Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle**
    - The control logic can be complex.
    - Lose instruction fetch and issue opportunity.
  - **No instruction is started in Cycle 6!**

- **Solution 2: Delay R-type’s Write by One Cycle**
  - **Delay R-type’s register write by one cycle:**
    - Now R-type instructions also use Reg File’s write port at Stage 5
    - Mem stage is a **NOOP** stage: nothing is being done.
**Modified Control & Datapath**

- **IR**: Instruction Register
- **PC**: Program Counter
- **A**: Register R[rs]
- **B**: Register R[rt]
- **M**: Memory
- **S**: Result of operation

**The Four Stages of Store**

- **Ifetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Calculate the memory address
- **Mem**: Write the data into the Data Memory

**The Three Stages of Beq**

- **Ifetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Compares the two register operands, selects the correct branch target address, and latches into PC

**Control Diagram**
Recall: Single cycle control!

The Main Control generates the control signals during Reg/Dec

- Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
- Control signals for Mem (MemWr Branch) are used 2 cycles later
- Control signals for Wr (MemtoReg MemWr) are used 3 cycles later

Data Stationary Control

Let’s Try it Out

10 lw r1, r2(35)
14 addI r2, r2, 3
20 sub r3, r4, r5
24 beq r6, r7, 100
30 ori r8, r9, 17
34 add r10, r11, r12
100 and r13, r14, 15

these addresses are octal
### Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10

- **Instruction (IR):** `lw r1, r2(35)`
- **Decode:** `lw r1`
- **Reg File:** `r2+3` (write-back)
- **Exec:** `lw r1`
- **Mem Ctrl:** `M[2×35]`
- **WB:** `r1=M[2×35]`
- **PC:** 30
- **Next PC:**

**Note Delayed Branch:** always execute `ori` after `beq`

### Fetch 100, Dcd 30, Ex 24, Mem 20, WB 14

- **Instruction (IR):** `lw r1, r2(35)`
- **Decode:** `lw r1`
- **Reg File:** `r2+3` (write-back)
- **Exec:** `lw r1`
- **Mem Ctrl:** `M[2×35]`
- **WB:** `r1=M[2×35]`
- **PC:** 100
- **Next PC:**

**Fill it in yourself!**

### Fetch 104, Dcd 100, Ex 30, Mem 24, WB 20

- **Instruction (IR):** `lw r1, r2(35)`
- **Decode:** `lw r1`
- **Reg File:** `r2+3` (write-back)
- **Exec:** `lw r1`
- **Mem Ctrl:** `M[2×35]`
- **WB:** `r1=M[2×35]`
- **PC:** 100
- **Next PC:**

**Fill it in yourself!**

### Fetch 110, Dcd 104, Ex 100, Mem 30, WB 24

- **Instruction (IR):** `lw r1, r2(35)`
- **Decode:** `lw r1`
- **Reg File:** `r2+3` (write-back)
- **Exec:** `lw r1`
- **Mem Ctrl:** `M[2×35]`
- **WB:** `r1=M[2×35]`
- **PC:** 100
- **Next PC:**

**Fill it in yourself!**
Recap: Data Hazards

° Avoid some “by design”
  • eliminate WAR by always fetching operands early (DCD) in pipe
  • eliminate WAW by doing all WBs in order (last stage, static)

° Detect and resolve remaining ones
  • stall or forward (if possible)

Hazards Detection

° Suppose instruction $i$ is about to be issued and a predecessor instruction $j$ is in the instruction pipeline.

New Inst $\rightarrow$ Inst $I$ $\rightarrow$ Inst $J$ $\rightarrow$ Window on execution:
Only pending instructions can cause exceptions

° A RAW hazard exists on register $\rho$ if $\rho \in \text{Rregs}(i) \cap \text{Wregs}(j)$
  • Keep a record of pending writes (for inst’s in the pipe) and compare
    with operand regs of current instruction.
  • When instruction issues, reserve its result register.
  • When on operation completes, remove its write reservation.

° A WAW hazard exists on register $\rho$ if $\rho \in \text{Wregs}(i) \cap \text{Wregs}(j)$

° A WAR hazard exists on register $\rho$ if $\rho \in \text{Wregs}(i) \cap \text{Rregs}(j)$
Record of Pending Writes In Pipeline Registers

- Current operand registers
- Pending writes
- hazard <=
  \[(rs == rw_{ex}) \land regW_{ex}) \lor (rs == rw_{mem}) \land regW_{mem}) \lor (rs == rw_{wb}) \land regW_{wb}) \lor (rt == rw_{ex}) \land regW_{ex}) \lor (rt == rw_{mem}) \land regW_{mem}) \lor (rt == rw_{wb}) \land regW_{wb})\]

Resolve RAW by “forwarding” (or bypassing)

- Detect nearest valid write op operand register and forward into op latches, bypassing remainder of the pipe
- Increase muxes to add paths from pipeline registers
- Data Forwarding = Data Bypassing

What about memory operations?

- If instructions are initiated in order and operations always occur in the same stage, there can be no hazards between memory operations!
- What does delaying WB on arithmetic operations cost?
  - cycles ?
  - hardware ?
- What about data dependence on loads?
  R1 <- R4 + R5
  R2 <- Mem[R2 + I]
  R3 <- R2 + R1
  \(\rightarrow \) “Delayed Loads”
- Can recognize this in decode stage and introduce bubble while stalling fetch stage (hint for lab 5!)
- Tricky situation:
  R1 <- Mem[R2 + I]
  Mem[R3+34] <- R1
  Handle with bypass in memory stage!

Compiler Avoiding Load Stalls:

<table>
<thead>
<tr>
<th>compiler</th>
<th>scheduled</th>
<th>unscheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>31%</td>
<td>54%</td>
</tr>
<tr>
<td>spice</td>
<td>14%</td>
<td>42%</td>
</tr>
<tr>
<td>tex</td>
<td>25%</td>
<td>65%</td>
</tr>
</tbody>
</table>
What about Interrupts, Traps, Faults?

- **External Interrupts:**
  - Allow pipeline to drain, Fill with NOPs
  - Load PC with interrupt address

- **Faults (within instruction, restartable):**
  - Force trap instruction into IF
  - disable writes till trap hits WB
  - must save multiple PCs or PC + state

- **Recall: Precise Exceptions ⇒ State of the machine is preserved as if program executed up to the offending instruction**
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations

---

Exception/Interrupts: Implementation questions

5 instructions, executing in 5 different pipeline stages!

- **Who caused the interrupt?**

  **Stage** | **Problem interrupts occurring**
  --- | ---
  IF | Page fault on instruction fetch; misaligned memory access; memory-protection violation
  ID | Undefined or illegal opcode
  EX | Arithmetic exception
  MEM | Page fault on data fetch; misaligned memory access; memory-protection violation; memory error

- **How do we stop the pipeline? How do we restart it?**
- **Do we interrupt immediately or wait?**
- **How do we sort all of this out to maintain preciseness?**

---

Exception Handling

- Use pipeline to sort this out!
  - Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don’t act on exception until it reaches WB stage

- Handle interrupts through “faulting noop” in IF stage

- When instruction reaches end of MEM stage:
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all instructions in earlier stages into noops!

---

Another look at the exception problem

- Program Flow

---

3/17/03 ©UCB Spring 2003 CS152 / Kubiatowicz Lec13.53

3/17/03 ©UCB Spring 2003 CS152 / Kubiatowicz Lec13.54

3/17/03 ©UCB Spring 2003 CS152 / Kubiatowicz Lec13.55

3/17/03 ©UCB Spring 2003 CS152 / Kubiatowicz Lec13.56
Resolution: Freeze above & Bubble Below

Flush accomplished by setting "invalid" bit in pipeline

FYI: MIPS R3000 clocking discipline

- 2-phase non-overlapping clocks
- Pipeline stage is two (level sensitive) latches

MIPS R3000 Instruction Pipeline

Recall: Data Hazard on r1

With MIPS R3000 pipeline, no need to forward from WB stage
MIPS R3000 Multicycle Operations

Ex: Multiply, Divide, Cache Miss

Use control word of local stage to step through multicycle operation

Stall all stages above multicycle operation in the pipeline

Drain (bubble) stages below it

Alternatively, launch multiply/divide to autonomous unit, only stall pipe if attempt to get result before ready

- This means stall mflo/mfhi in decode stage if multiply/divide still executing
- Extra credit in Lab 5 does this

Is CPI = 1 for our pipeline?

- Remember that CPI is an “Average # cycles/inst

- CPI here is 1, since the average throughput is 1 instruction every cycle.

- What if there are stalls or multi-cycle execution?

- Usually CPI > 1. How close can we get to 1??

Summary

- 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
  - Control: early evaluation & PC, delayed branch, prediction

- Data hazards must be handled carefully:
  - RAW data hazards handled by forwarding
  - WAW and WAR hazards don’t exist in 5-stage pipeline

- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)

- Exceptions in 5-stage pipeline recorded when they occur, but acted on only at WB (end of MEM) stage
  - Must flush all previous instructions