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>Time</th>
<th>Task Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 PM</td>
<td>A B C D</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>B C D</td>
</tr>
<tr>
<td>9</td>
<td>C D</td>
</tr>
<tr>
<td></td>
<td>D</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
- Administrative
- Hazards/Forwarding
- Exceptions
- Review MIPS R3000 pipeline

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

Time (clock cycles)

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 \((1 \times .5 + 2 \times .5) = 1.5\)
  * Total CPI might then be: \(1.5 \times .2 + 1 \times .8 = 1.1\) (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

```
Time (clock cycles)
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or r8,r1,r9
xor r10,r1,r11
```

Data Hazard Solution: Forwarding

- “Forward” result from one stage to another

```
Time (clock cycles)
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or r8,r1,r9
xor r10,r1,r11
```

Forwarding (or Bypassing): What about Loads?

- Dependencies backwards in time are hazards

```
Time (clock cycles)
lw r1,0(r2)
sub r4,r1,r3
```

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

```
Time (clock cycles)
lw r1,0(r2)
sub r4,r1,r3
```

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

Pipelined Processor (almost) for slides

Pipelined Datapath (as in book): hard to read

- What happens if we start a new instruction every cycle?
Test was way too hard! Don’t go away!
- It is graded on a curve (only 15% of grade as well)
- Average: 57.9, Standard Dev: 10.8

CS152 Midterm I results

Tomorrow: Sections in Cory lab (119)
- Since we are going to be testing things that you may not have tried, think again carefully about testing your design!
- Perhaps read the paper by Doug Clark on handouts page
- Everyone should go to section

Problems with memory on Lab 4????

Get started on LAB 5!
- Problem 0 due tomorrow night at 12 Midnight via email
  - Evaluate your teammates.
  - Organization on Lab due Thursday by Midnight via email

The Big Picture: Where are We Now?

The Five Classic Components of a Computer

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

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

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

Pipelining the R-type and Load Instruction

° 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.

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{Cycle} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\text{R-type} & \text{Ifetch} & \text{Reg/Dec} & \text{Exec} & \text{Mem} & \text{Wr} & \text{Mem} & \text{Wr} \\
\end{array}
\]

Modified Control & Datapath

\[
\begin{align*}
\text{IR} & \leftarrow \text{Mem}[\text{PC}]; \text{PC} \leftarrow \text{PC}+4; \\
A & \leftarrow R[rs]; B \leftarrow R[rt]; \\
S & \leftarrow A + B; \\
R[rd] & \leftarrow M; \\
S & \leftarrow A + SX; \\
M & \leftarrow \text{Mem}[S]; \\
\text{Mem}[S] & \leftarrow B; \\
R[rd] & \leftarrow M; \\
R[rt] & \leftarrow M; \\
\end{align*}
\]

The Four Stages of Store

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{Cycle} & 1 & 2 & 3 & 4 \\
\hline
\text{Store} & \text{Ifetch} & \text{Reg/Dec} & \text{Exec} & \text{Mem} & \text{Wr} & \text{Wr} \\
\end{array}
\]

The Three Stages of Beq

\[
\begin{align*}
\text{Ifetch} : \text{Instruction Fetch} \\
& \quad \text{Fetch the instruction from the Instruction Memory} \\
\text{Reg/Dec} : \text{Registers Fetch and Instruction Decode} \\
\text{Exec} : \text{Calculate the memory address} \\
\text{Mem} : \text{Write the data into the Data Memory} \\
\end{align*}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
\text{Cycle} & 1 & 2 & 3 & 4 \\
\hline
\text{Beq} & \text{Ifetch} & \text{Reg/Dec} & \text{Exec} & \text{Mem} \\
\end{array}
\]
Control Diagram

IR <- Mem[PC]; PC <- PC + 4
A <- R[rs]; B <- R[rt]
S <- A + B;
S <- A or ZX;
S <- A + SX;
S <- A + SX;

M <- S
M <- Mem[S]
Mem[S] <- B
R[rd] <- S;
R[rt] <- S;
R[rd] <- M;

S <- A or ZX;
S <- A + SX;
M <- Mem[S]
R[rt] <- S;

S <- A + SX;
Mem[S] <- B

Data Stationary 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

Datapath + Data Stationary Control

Let's Try it Out

10  lw   r1, r2(35)
14  addl  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 114, Dcd 110, Ex 104, Mem 100, WB 30**

- **Reg File**
- **Exec**
- **Mem Ctrl**
- **WB Ctrl**
- **Mem**
- **Data Mem**
- **IR**
- **Inst Mem**
- **Decode**
- **Reg File**
- **Next PC**
- **PC**

Fill in yourself!

**Pipeline Hazards Again**

<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**

RAW (read after write) Data Hazard

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

WAW (write after write) Data Hazard

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

WAR (write after read) Data Hazard

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

**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)

**Hazard Detection**

- Suppose instruction \( i \) is about to be issued and a predecessor instruction \( j \) is in the instruction pipeline.
- 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 \( \rho \in \text{Wregs}(i) \cap \text{Rregs}(j) \)
Lec13.49

Record of Pending Writes

- Current operand registers
- Pending writes
- hazard <=
  \[((rs == rw ex) & regW ex) OR ((rs == rw mem) & regW mem) OR ((rs == rw wb) & regW wb) OR ((rt == rw ex) & regW ex) OR ((rt == rw mem) & regW mem) OR ((rt == rw wb) & regW wb)\]

Lec13.50

Resolve RAW by forwarding

- 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

Lec13.51

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 + 1 ]
  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 + 1 ]
  Mem[R3+34] <- R1
  Handle with bypass in memory stage!

Lec13.52

Compiler Avoiding Load Stalls:

<table>
<thead>
<tr>
<th>Compiler</th>
<th>scheduled</th>
<th>unscheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>25%</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>

% loads stalling pipeline
What about Interrupts, Traps, Faults?

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

- **Fa ults (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 Problem

- **Exceptions/Interrupts:** 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - Restart?
  - 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
  - Load with data page fault, Add with instruction page fault?
  - Solution 1: interrupt vector/instruction 2: interrupt ASAP, restart everything incomplete

---

Another look at the exception problem

- **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 reache WB stage

- **Handle interrupts through “faulting noop” in IF stage**
- **When instruction reaches WB stage:**
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all instructions in earlier stages into noops!

---

Exception Handling

- **Reg**
  - **alu**
  - **d mem**
  - **wb**
  - **mem**

- **IAU**
  - **npc**
  - **mem**

- **detect bad instruction address**
- **detect bad instruction**
- **detect overflow**
- **detect bad data address**

- Allow exception to take effect
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

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Decode</th>
<th>ALU / E.A</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E.A.</td>
<td>TLB</td>
<td>D-Cache</td>
</tr>
</tbody>
</table>

Recall: Data Hazard on r1

- Write in phase 1, read in phase 2 => eliminates bypass from WB

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

Ex: Multiply, Divide, Cache Miss

Stall all stages above multicycle operation in the pipeline

Drain (bubble) stages below it

Use control word of local stage state to step through multicycle operation

Issues in Pipelined design

- Pipelining
  - Issue one instruction per (fast) cycle
  - ALU takes multiple cycles
  - Issue rate, FU stalls, FU depth

- Super-pipeline
  - Issue multiple scalar instructions per cycle
  - Clock skew, FU stalls, FU depth

- Super-scalar
  - Issue multiple scalar operations
  - Hazard resolution

- VLIW (“EPIC”)
  - Each instruction specifies multiple scalar operations
  - Packing

- Vector operations
  - Each instruction specifies series of identical operations
  - Applicability

Summary #1/2: Pipelining

- What makes it easy
  - All instructions are the same length
  - Just a few instruction formats
  - Memory operands appear only in loads and stores

- What makes it hard? HAZARDS!
  - Structural hazards: suppose we had only one memory
  - Control hazards: need to worry about branch instructions
  - Data hazards: an instruction depends on a previous instruction

- Pipelines pass control information down the pipe just as data moves down pipe

Summary #2/2

- Pipelines pass control information down the pipe just as data moves down pipe
- Forwarding/Stalls handled by local control
- Exceptions stop the pipeline
- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
- More performance from deeper pipelines, parallelism