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!

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 Five Classic Components of a Computer

Today’s Topics:
- Recap last lecture/finish datapath
- Pipelined Control/ Do it yourself  
- Administrivia
- 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

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

Detection is easy in this case! (right half highlight means read, left half write)
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 \((1 \times 0.5 + 2 \times 0.5) = 1.5\)
  • Total CPI might then be: \(1.5 \times 0.2 + 1 \times 0.8 = 1.1\) (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

```
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

```
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

```
lw r1,0(r2)
sub r4,r1,r3
```

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

Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards

```
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

IR ← Mem[PC]; PC ← PC+4;
A ← R[rs]; B ← R[rt]
S ← A + B;
S ← A + SX;
S ← A + SX;
If Cond
PC ← PC+5X;
M ← Mem[S]; Mem[S] ← B
R[rd] ← S;
R[rd] ← M;
R[rd] ← M;

Pipelined Processor (almost) for slides

° What happens if we start a new instruction every cycle?
**Administrivia**

- Get started on LAB 5!
  - Problem 0 due tonight at 12 Midnight via email: evaluate your teammates.
  - Organization on Lab due by Monday at 5pm
- Next week: Sections in Cory lab. Run "mystery program" on Lab 4 to test your design.
  - 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

**The Big Picture: Where are We Now?**

- The Five Classic Components of a Computer
- The Four Stages of R-type

**Pipelining the Load Instruction**

<table>
<thead>
<tr>
<th>Clock</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>1st lw</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
</tr>
<tr>
<td>2nd lw</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
</tr>
<tr>
<td>3rd lw</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
</tr>
</tbody>
</table>

- 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

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.
- 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
- 2 ways to solve this pipeline hazard.
Modified Control & Datapath

IR <- Mem[PC]; PC <- PC+4;
A <- R[rs]; B <- R[rt];
S <- A + B;
M <- S;
R[rd] <- M;
S <- A + SX;
Mem[S] <- B;
if Cond PC < PC+sx;

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 operand,
  • select correct branch target address
  • latch into PC

Control Diagram
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

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 14, Decode 10

Fetch 20, Decode 14, Exec 10

Fetch 24, Decode 20, Exec 14, Mem 10

Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10

Note Delayed Branch: always execute ori after beq
### 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Structural Hazard**

- IFetch
- DCD

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

**Control Hazard**

- IFetch
- DCD

### 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)

### Data Hazards

#### RAW (read after write) Data Hazard

- IF\, DCD\, EX\, Mem\, WB

#### WAW (write after write) Data Hazard

- IF\, DCD\, EX\, Mem\, WB

#### WAR (write after read) Data Hazard

- IF\, DCD\, OF\, Ex\, Mem

### 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)

### 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 \( \rho \in \text{Wregs}(i) \cap \text{Rregs}(j) \)
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

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?
  - \( R1 \leftarrow R4 + R5 \)
  - \( R2 \leftarrow \text{Mem}[R2 + I] \)
  - \( R3 \leftarrow R2 + R1 \)
  - "Delayed Loads"
- Can recognize this in decode stage and introduce bubble while stalling fetch stage (hint for lab 5!)
- Tricky situation:
  - \( R1 \leftarrow \text{Mem}[R2 + I] \)
  - \( \text{Mem}[R3+34] \leftarrow R1 \)
  - Handle with bypass in memory stage!

What about interrupts, traps, faults?
- External Interrupts:
  - Allow pipeline to drain,
  - 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

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>

% loads stalling pipeline
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

- 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
- Freeze accomplished by setting

Program Flow

Time

Data TLB
IFetch Dcd Exec Mem WB
Bad Inst
IFetch Dcd Exec Mem WB
Inst TLB fault
IFetch Dcd Exec Mem WB
Overflow
IFetch Dcd Exec Mem WB

npc
I mem
Regs
alu
S
D mem
m
IAU

10/13/99 ©UCB Fall 1999
CS152 / Kubiatowicz
Lec13.55
**FYI: MIPS R3000 clocking discipline**

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

![Diagram of 2-phase non-overlapping clocks and pipeline stages]

**MIPS R3000 Instruction Pipeline**

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

**Resource Usage**

- TLB
- I-cache
- RF
- ALU
- D-Cache

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

**Recall: Data Hazard on r1**

<table>
<thead>
<tr>
<th>Operation</th>
<th>IF</th>
<th>ID/Reg</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td>Im</td>
<td>Reg</td>
<td>Dm</td>
<td>Reg</td>
<td>Dm</td>
</tr>
<tr>
<td>sub r4, r1, r3</td>
<td>Im</td>
<td>Reg</td>
<td>Dm</td>
<td>Reg</td>
<td>Dm</td>
</tr>
<tr>
<td>and r6, r1, r7</td>
<td>Im</td>
<td>Reg</td>
<td>Dm</td>
<td>Reg</td>
<td>Dm</td>
</tr>
<tr>
<td>or r8, r1, r9</td>
<td>Im</td>
<td>Reg</td>
<td>Dm</td>
<td>Reg</td>
<td>Dm</td>
</tr>
<tr>
<td>xor r10, r1, r11</td>
<td>Im</td>
<td>Reg</td>
<td>Dm</td>
<td>Reg</td>
<td>Dm</td>
</tr>
</tbody>
</table>

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

**MIPS R3000 Multicycle Operations**

With MIPS R3000 pipeline, no need to forward from WB stage
### Issues in Pipelined design

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipelining</td>
<td>Issue one instruction per (fast) cycle</td>
<td>Issue rate, FU stalls, FU depth</td>
</tr>
<tr>
<td>Super-pipeline</td>
<td>ALU takes multiple cycles</td>
<td>Clock skew, FU stalls, FU depth</td>
</tr>
<tr>
<td>Super-scalar</td>
<td>Issue multiple scalar instructions per cycle</td>
<td>Hazard resolution</td>
</tr>
<tr>
<td>VLIW (“EPIC”)</td>
<td>Each instruction specifies multiple scalar operations</td>
<td>Packing</td>
</tr>
<tr>
<td>Vector operations</td>
<td>Each instruction specifies series of identical operations</td>
<td>Applicability</td>
</tr>
</tbody>
</table>

### 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
- Forwarding/Stalls handled by local control
- Exceptions stop the pipeline

### 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