CS152
Computer Architecture and Engineering
Lecture 13

Introduction to Pipelining II:
Control

March 15, 1999
John Kubiatowicz (http.cs.berkeley.edu/~kubitron)

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/
Recap: Sequential Laundry

Sequential laundry takes 8 hours for 4 loads

If they learned pipelining, how long would laundry take?
Recap: Pipelining Lessons (its intuitive!)

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- 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 $\leq$ Number of stages

Speedup $\leq$ Time for unpipelined operation

Time for longest stage

Example: 40ns data path, 5 stages, Longest stage is 10 ns, Speedup $\leq$ 4
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., multiple memory accesses, multiple register writes
  - solutions: multiple memories, stretch pipeline
- **control hazards**: attempt to make a decision before condition is evaluated
  - e.g., any conditional branch
  - solutions: prediction, delayed branch
- **data hazards**: attempt to use item before it is ready
  - e.g., add r1, r2, r3; sub r4, r1, r5; lw r6, 0(r7); or r8, r6, r9
  - solutions: forwarding/bypassing, stall/bubble
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
  - Advanced Pipelining?
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 or ZX;
S ← A + SX;
S ← A + SX;

If Cond
PC < PC+SX;

M ← Mem[S]
Mem[S] ← B

R[rd] ← S;
R[rt] ← S;
R[rd] ← M;

Equal

Reg. File

Mem Access

Data Mem

Exec

Reg File

Mem

IR

Inst. Mem

Reg File

A

B

S

D

M

PC

Next PC
What happens if we start a new instruction every cycle?
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
We have pipeline conflict or structural hazard:
  • Two instructions try to write to the register file at the same time!
  • Only one write port
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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
</tr>
</tbody>
</table>

  • R-type uses Register File’s Write Port during its 4th stage

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Wr</td>
</tr>
</tbody>
</table>

° 2 ways to solve this pipeline hazard.
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.

Clock cycle 1-9:

<table>
<thead>
<tr>
<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>
<th>Cycle 8</th>
<th>Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td>Mem</td>
<td>Wr</td>
<td>Mem</td>
<td>Wr</td>
</tr>
<tr>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

©UCB Spring 1999 CS152 / Kubiatowicz

Lec13.14
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 or ZX;
M ← S
R[rt] ← M;

S ← A + SX;
M ← Mem[S]
R[rd] ← M;

S ← A + SX;
Mem[S] ← B

if Cond PC < PC+SX;

Reg. File

M <- Equal

Mem Access

Data Mem

Exec

Reg. File

IR

Inst. Mem

Next PC

PC

3/15/99
©UCB Spring 1999
Lec13.15
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

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;
If Cond PC < PC+SX;

M <- S
M <- S
M <- Mem[S]
Mem[S] <- B

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

Equal

Reg. File

Mem Access

Mem

Data
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
 Administrivia

° Get started on LAB 5!
  • Problem 0 due tonight at 12 Midnight via email: evaluate your teammates.
  • Organization on Lab due by Wednesday.

° Starting tomorrow: Sections in Cory lab. Tomorrow: run “mystery program” on Lab 4.

° Generally positive feedback about course.
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
Start: Fetch 10

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
Fetch 14, Decode 10

```
10 lw r1, r2(35)
14 add 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
```
Fetch 20, Decode 14, Exec 10
Fetch 24, Decode 20, Exec 14, Mem 10

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

Note Delayed Branch: always execute ori after beq
Fetch 100, Dcd 30, Ex 24, Mem 20, WB 14

```
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
```
Fill it in yourself!

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

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

Fill it in yourself!

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

PC

©UCB Spring 1999
Fetch 114, Dcd 110, Ex 104, Mem 100, WB 30

Fill it in yourself!

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

©UCB Spring 1999
Pipeline Hazards Again

- **Structural Hazard**:
  - IFetch → DCD

- **Control Hazard**:
  - IFetch → DCD

- **RAW (read after write) Data Hazard**:
  - IF → DCD → EX → Mem → WB

- **WAW Data Hazard (write after write)**:
  - IF → DCD → EX → Mem → WB

- **WAR Data Hazard (write after read)**:
  - IF → DCD → OF → Ex → RS
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 \( \rho \in Rregs( i ) \cap Wregs( j ) \)
  - 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 \( \rho \in Wregs( i ) \cap Wregs( j ) \)

- A WAR hazard exists on register \( \rho \) if \( \rho \in Wregs( i ) \cap Rregs( j ) \)
Record of Pending Writes

- Current operand registers
- Pending writes
- hazard <=

\[ ((rs == rw_{ex}) \ & \ regW_{ex}) \ OR \]
\[ ((rs == rw_{mem}) \ & \ regW_{me}) \ OR \]
\[ ((rs == rw_{wb}) \ & \ regW_{wb}) \ OR \]
\[ ((rt == rw_{ex}) \ & \ regW_{ex}) \ OR \]
\[ ((rt == rw_{mem}) \ & \ regW_{me}) \ OR \]
\[ ((rt == rw_{wb}) \ & \ regW_{wb}) \]
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 <- R4 + R5
  R2 <- Mem[ R2 + I ]
  R3 <- R2 + R1

=> "Delayed Loads"

Tricky situation:
  R1 <- Mem[ R2 + I ]
  Mem[R3+34] <- R1
Compiler Avoiding Load Stalls:

% loads stalling pipeline

<table>
<thead>
<tr>
<th>Program</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>

©UCB Spring 1999
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

Refer to MIPS solution
Exception Handling

- Detect bad instruction address
- Detect bad instruction
- Detect overflow
- Detect bad data address

IAU → npc → l mem → lw $2,20($5) → PC

B → A → im → n op → rw → I → D mem → m → Regs

Allow exception to take effect
## 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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Problem interrupts occurring</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory-protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory-protection violation; memory error</td>
</tr>
</tbody>
</table>

- Load with data page fault, Add with instruction page fault?
- Solution 1: interrupt vector/instruction 2: interrupt ASAP, restart everything incomplete
Resolution: Freeze above & Bubble Below

- IAU
- npc
- I mem
- op rw rs rt
- PC
- freeze
- bubble
- alu
- D mem
- m
- Regs
- B A
- im
- n op lw
- n op lw
- n op lw
- n op lw
- n op lw
- n op lw
FYI: MIPS R3000 clocking discipline

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

Edge-triggered

phi1

phi2
### MIPS R3000 Instruction Pipeline

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Decode Reg. Read</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>TLB</td>
<td>I-cache</td>
<td>RF</td>
<td>E.A.</td>
<td>TLB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>D-Cache</td>
<td></td>
</tr>
</tbody>
</table>

#### Resource Usage

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

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

- 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

° Super-pipeline
   - Issue one instruction per (fast) cycle
   - ALU takes multiple cycles

° Super-scalar
   - Issue multiple scalar instructions per cycle

° VLIW ("EPIC")
   - Each instruction specifies multiple scalar operations
   - Compiler determines parallelism

° Vector operations
   - Each instruction specifies series of identical operations

Limitation
   Issue rate, FU stalls, FU depth

Clock skew, FU stalls, FU depth

Limitation
   Issue rate, FU stalls, FU depth

Hazard resolution

Packing

Applicability
Historical Perspective

- **80’s RISC pipelines** (mips, sparc, ...)
- **Load/Store ISA** (cdc 6600, 7600, Cray-1, ...)
- **Cache** (ibm 360/85, ...)
- **Virtual Memory** (multics, ge-645, ibm 360/67, ...)
- **Dynamic Inst. Scheduling with extensive pipelining** (ibm 360/91)
- **Inst. Buffering (Stretch - 100x ibm704)**
- **Microprogramming**

- **1966**
  - 60ns hardwired
  - 8x16b bus
  - 780ns mem

- **1967**
  - 25x basic model

- **1961**
  - 100x ibm704

- **Today**
  - 80ns, 2Kb Ctrl. St
  - 4x16b bus
  - 960ns mem
  - 32KB cache
  - 60-160ns
Technology Perspective
Partitioned Instruction Issue (simple Superscalar)

independent int and FP issue to separate pipelines

Single Issue Total Time = Int Time + FP Time

Max Speedup: \[
\frac{\text{Total Time}}{\text{MAX(Int Time, FP Time)}}
\]
Example: DAXPY

Basic Loop: <- Rm+Ry

Total Single Issue Cycles: 19 (7 integer, 12 floating point)
Minimum with Dual Issue: 12
Potential Speedup: 1.6 !!!

Actual Cycles: 18
Unrolling

Basic Loop:

<table>
<thead>
<tr>
<th>Load</th>
<th>a &lt;- Ai</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>y &lt;- Yi</td>
</tr>
<tr>
<td>Mult</td>
<td>m &lt;- a*s</td>
</tr>
<tr>
<td>Add</td>
<td>r &lt;- m+y</td>
</tr>
<tr>
<td>Store</td>
<td>Ai &lt;- r</td>
</tr>
<tr>
<td>Inc</td>
<td>Ai</td>
</tr>
<tr>
<td>Inc</td>
<td>Yi</td>
</tr>
<tr>
<td>Dec</td>
<td>i</td>
</tr>
<tr>
<td>Branch</td>
<td></td>
</tr>
</tbody>
</table>

about 9 inst. per 2 FP ops

Unrolled Loop:

| Load, load, mult, add, store |
| Load, load, mult, add, store |
| Load, load, mult, add, store |
| Load, load, mult, add, store |
| Inc, Inc, dec, branch |
| Inc, Inc, dec, branch |

about 6 inst. per 2 FP ops dependencies between instructions remain.

Reordered Unrolled Loop:

| Load, load, load, . . . |
| Mult, mult, mult, mult, |
| Add, add, add, add, add, |
| Store, store, store, store |
| Inc, Inc, dec, branch |

schedule 24 inst basic block relative to pipeline
- delay slots
- function unit stalls
- multiple function units
- pipeline depth
### Software Pipelining

**Table: Pipelining Example**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>load</code></td>
<td><code>a &lt;- A1</code></td>
</tr>
<tr>
<td><code>load</code></td>
<td><code>y &lt;- Y1</code></td>
</tr>
<tr>
<td><code>mult</code></td>
<td><code>m &lt;- a*s</code></td>
</tr>
<tr>
<td><code>add</code></td>
<td><code>r &lt;- m+y</code></td>
</tr>
<tr>
<td><code>inc</code>, <code>dec</code></td>
<td><code>m &lt;- a's</code></td>
</tr>
<tr>
<td><code>store</code></td>
<td><code>Ai &lt;- r</code></td>
</tr>
<tr>
<td><code>branch</code></td>
<td><code>inc</code></td>
</tr>
</tbody>
</table>

**Loop:**

- `load` `a'' <- Ai+3`
- `load` `y'' <- Yi+2`
- `mult` `m'' <- a''*s`
- `add` `r' <- m''+y'`
- `store` `Ai <- r`
- `inc` `Ai+3`
- `inc` `Yi`
- `dec` `i`
- `a'' <- a'''; Y'' <- y''; m'' <- m''; r' <- r'`  
- `branch`

---

**Pipelined Loop:**

- `load` `a'''' <- Ai+3`
- `load` `y'''' <- Yi+2`
- `mult` `m'''' <- a''''*s`
- `add` `r'' <- m''''+y'''
- `store` `Ai+1 <- r'`
- `add` `r'''' <- m''''+y'''
- `inc` `Ai+3`
- `inc` `Yi+2`
Multiple Pipes/ Harder Superscalar

Issues:
- Reg. File ports
- Detecting Data Dependences
- Bypassing
- RAW Hazard
- WAR Hazard
- Multiple load/store ops?
- Branches
Branch penalties in superscalar

Example: resolved in op-fetch stage, single exposed delay (ala MIPS, Sparc)

<table>
<thead>
<tr>
<th>I-fetch</th>
<th>Branch</th>
<th>delay</th>
</tr>
</thead>
</table>

Squash 2

<table>
<thead>
<tr>
<th>I-fetch</th>
<th>Branch</th>
<th>delay</th>
</tr>
</thead>
</table>

Squash 1
Summary: 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?
  • 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

° 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