CS152: Computer Architecture and Engineering
Introduction to Pipelining

October 22, 1997
Dave Patterson (http.cs.berkeley.edu/~patterson)

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

**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
Recap: Graphically Representing Pipelines

- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths
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 \texttt{r1}, \texttt{r2}, \texttt{r3}; sub \texttt{r4}, \texttt{r1}, \texttt{r5}; lw \texttt{r6}, 0(\texttt{r7}); or \texttt{r8}, \texttt{r6}, \texttt{r9}
  - solutions: forwarding/bypassing, stall/bubble
Recap: Pipelined Datapath with Data Stationary Control

Just like Time-State!

Operand Register Selects

ALU Op

MEM Op

Result Reg Select and Enable

<= PC + 4 + immed
Recap

° Pipelining is a fundamental concept
  • multiple steps using distinct resources

° Utilize capabilities of the Datapath by pipelined instruction processing
  • start next instruction while working on the current one
  • limited by length of longest stage (plus fill/flush)
  • detect and resolve hazards

° What makes it easy
  • all instructions are the same length
  • just a few instruction formats
  • memory operands appear only in loads and stores

° Hazards make it hard

° We’ll build a simple pipeline and look at these issues
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?
Recap: Control Diagram

IR ← Mem[PC]; PC ← PC+4;

A ← R[rs]; B ← R[rt]

S ← A + B;

S ← A + SX;

S ← A + SX;

S ← A or ZX;

S ← A + SX;

M ← Mem[S]

M ← S

M ← S

Mem[S] ← B

M ← S

M ← S

R[rd] ← S;

R[rt] ← S;

R[rd] ← M;

If Cond PC < PC+SX;

Exec

Reg. File

Mem Access

Data Mem

M

Equal

Next PC

PC

IR

Inst. Mem

Reg File

A

B

S

D

M

cs 152  L1 3 .10

DAP Fa97, © U.CB
But recall use of “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

- Instruction Memory (Inst. Mem)
- Instruction Register (IR)
- Functional Unit (fun)
- Register File (Reg. File)
- Decode
- Memory Control (Mem Ctrl)
- Memory Access
- Write Back (WB Ctrl)
- Register File (Reg. File)
- Next PC
- PC
- Immediate (im)
- Address (A, B)
- Data Memory (Data Mem)
- RS, RT
- V, RW, WB, ME, EX
- M
- S
- D
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
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

IR

Decide 14

Reg File

Mem Access

WB Ctrl

Inst. Mem

IF

ID

Mem Ctrl

PC

Next PC

Mem

Data Mem

Add

Sub

Beq

Ori

Add

And

lw r1, r2(35)

addI r2, r2, 3

sub r3, r4, r5

beq r6, r7, 100

ori r8, r9, 17

add r10, r11, r12

and r13, r14, 15
Fetch 20, Decode 14, Exec 10

IR

Inst. Mem

addl r2, r2, 3

Decide

lwr1

Reg File

r2

Exec

M

Mem Access

Data Mem

WB

Reg. File

EX

10 lw r1, r2(35)

ID

14 addl r2, r2, 3

IF

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
Administrative Issues

° Schedule Ahead

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- pipeline (5)
- cache (6)
- xtra & writeup

midterm

° Course Feedback

- Like on-line lecture notes!! pace of class!!
- Like Computers in the news!!
- Prerequisite Quiz? 39 great, 2 so-so, 1 bad idea
- Online Submission?
- Spread TA office hours?
- Slow lectures last 20 minutes?

° Computers in the news:

- Alpha/Intel patent scabble to be settled this week?
Note Delayed Branch: always execute \texttt{ori} after \texttt{beq}
Fetch 100, Dcd 30, Ex 24, Mem 20, WB 14

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

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
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
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
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**

<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>EX</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>EX</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>

**WAW Data Hazard** (write after write)

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

**WAR Data Hazard** (write after read)
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 \( \rho \in \text{Wregs}(i) \cap \text{Rregs}(j) \)
Record of Pending Writes

- Current operand registers
- Pending writes
- hazard <=

\[
\text{hazard} \leq \left( (\text{rs} == \text{rw}_{\text{ex}}) \land \text{regW}_{\text{ex}} \right) \lor \left( (\text{rs} == \text{rw}_{\text{mem}}) \land \text{regW}_{\text{me}} \right) \lor \left( (\text{rs} == \text{rw}_{\text{wb}}) \land \text{regW}_{\text{wb}} \right) \lor \left( (\text{rt} == \text{rw}_{\text{ex}}) \land \text{regW}_{\text{ex}} \right) \lor \left( (\text{rt} == \text{rw}_{\text{mem}}) \land \text{regW}_{\text{me}} \right) \lor \left( (\text{rt} == \text{rw}_{\text{wb}}) \land \text{regW}_{\text{wb}} \right)
\]
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"
Compiler Avoiding Load Stalls:

% loads stalling pipeline

- gcc: scheduled 31%, unscheduled 54%
- spice: scheduled 14%, unscheduled 42%
- tex: scheduled 25%, unscheduled 65%
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

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**
- **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, check last stage
  - Solution 2: interrupt ASAP, restart everything incomplete
Resolution: Freeze above & Bubble Below
FYI: MIPS R3000 clocking discipline

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

Edge-triggered

```
phi1

phi2
```

```
phi1
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></td>
<td></td>
<td></td>
<td>E.A.</td>
<td>TLB</td>
</tr>
</tbody>
</table>

Resource Usage

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

\[
\text{mul} \quad \text{Rd} \rightarrow \text{R} \rightarrow \text{T} \rightarrow \text{reg file}
\]
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
Hazard resolution
Packing
Applicability
Historical Perspective

- **Load/Store ISA** (cdc 6600, 7600, Cray-1, ...)
  - 1966
  - 60ns hardwired 8x16b bus 780ns mem

- **80's RISC pipelines** (mips, sparc, ...)
  - Dynamic Inst. Scheduling with extensive pipelining (ibm 360/91)
    - 25x basic model
    - 1967
  - Inst. Pipelining Inst. Buffering (Stretch - 100x ibm704)
    - 1961

- **Virtual Memory** (multics, ge-645, ibm 360/67, ...)
  - TLB

- **Cache** (ibm 360/85, ...)
  - 80ns, 2Kb Ctrl. St 4x16b bus 960ns mem 32KB cache 60-160ns

- **Early 90's RISC Superscalars**

- **Today**
Technology Perspective

- Transistors
- Year

- i4004
- i8080
- i8086
- i80286
- i80386
- i80486
- Pentium

- 4 bit
- 8 bit
- 16 bit
- 32 bit
- 64 bit
- Superscalar
Partitioned Instruction Issue (simple Superscalar)

independent int and FP issue to separate pipelines

Single Issue Total Time = Int Time + FP Time

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

<table>
<thead>
<tr>
<th>Basic Loop:</th>
<th>Cycles</th>
<th>Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>load Ra &lt;- Ai</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>load Ry &lt;- Yi</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>fmult Rm &lt;- Ra*Rx</td>
<td>1+6</td>
<td>6 cycle mult, 3 stage</td>
</tr>
<tr>
<td>fadd Rs &lt;- Rm+Ry</td>
<td>1+4</td>
<td>4 cycle add, 2 stage</td>
</tr>
<tr>
<td>store Ai &lt;- Rs</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>inc Yi</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dec i</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>inc Ai</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>branch</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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:
- load a ← Ai
- load y ← Yi
- mult m ← a*s
- add r ← m+y
- store Ai ← r
- inc Ai
- inc Yi
- dec i
- branch

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

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,
- 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

| load a <- A1 | load a' <- A2 |
| load y <- Y1 | load y' <- Y2 |
| mult m <- a*s | mult m' <- a'*s |
| addr <- m+y | add r' <- m'+y' |
| inc, dec | inc, dec |
| storeAi <- r | store Ai+1 <- r' |
| branch | add r''<-m''+y'' |
| | inc |

Pipelined 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
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></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>delay</td>
<td></td>
</tr>
</tbody>
</table>

Squash 2

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

Squash 1
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