I come back to the same thing: We've got the greatest pipeline in the company's history in the next 12 months, and we've had the most amazing financial results possible over the last five years, and we're predicting being back at double-digit revenue growth in fiscal year '06.  

- Steve Ballmer (Microsoft)
CS152
Computer Architecture and Engineering
Lecture 10

Introduction to Pipelining

March 1, 2004
John Kubiatowicz (www.cs.berkeley.edu/~kubitron)

lecture slides: http://inst.eecs.berkeley.edu/~cs152/
The Big Picture: Where are We Now?

° The Five Classic Components of a Computer

° Next Topics:
  • Pipelining by Analogy
  • Pipeline hazards
Pipelining is Natural!

° Laundry Example
° Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
° Washer takes 30 minutes
° Dryer takes 40 minutes
° “Folder” takes 20 minutes
Sequential laundry takes 6 hours for 4 loads

If they learned pipelining, how long would laundry take?
Pipelined Laundry: Start work ASAP

Pipelined laundry takes 3.5 hours for 4 loads
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.
The Five Stages of Load

- **Ifetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Calculate the memory address
- **Mem**: Read the data from the Data Memory
- **Wr**: Write the data back to the register file
Note: These 5 stages were there all along!

IR <= MEM[PC]  
PC <= PC + 4  

0000

ALUout <= PC +SX  

0001

ALUout <= A fun B  
0100

ALUout <= A op ZX  
0110

ALUout <= A + SX  
1000

ALUout <= A + SX  
1011

If A = B then PC <= ALUout  
0010

R-type

ORi

LW

SW

BEQ

ALUout <= PC +SX  

0101

ALUout <= ALUout  
0111

ALUout <= M  
1010

MEM[ALUout] <= B  
1100

R[rd] <= ALUout  
R[rt] <= M  
R[rt] <= M  

©UCB Spring 2004
Pipelining

- Improve performance by increasing throughput

Ideal speedup is number of stages in the pipeline. Do we achieve this?
Pipelining

° Improve performance by increasing throughput

**Single-cycle** ($T_c = 800\text{ps}$)

**Pipelined** ($T_c = 200\text{ps}$)

**Ideal speedup is number of stages in the pipeline.**

**Do we achieve this?**
Basic Idea

What do we need to add to split the datapath into stages?
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
Conventional Pipelined Execution Representation

Time

Program Flow

IFetch | Dcd | Exec | Mem | WB

IFetch | Dcd | Exec | Mem | WB

IFetch | Dcd | Exec | Mem | WB

IFetch | Dcd | Exec | Mem | WB

IFetch | Dcd | Exec | Mem | WB

IFetch | Dcd | Exec | Mem | WB

IFetch | Dcd | Exec | Mem | WB
Single Cycle, Multiple Cycle, vs. Pipeline

Single Cycle Implementation:

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Store</td>
<td>Waste</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Multiple Cycle Implementation:

<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>
<th>Cycle 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Store</td>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ifetch</td>
<td>Reg</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td>Ifetch</td>
<td>Reg</td>
<td>Exec</td>
<td>Mem</td>
<td>Ifetch</td>
</tr>
</tbody>
</table>

Pipeline Implementation:

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Store</td>
</tr>
<tr>
<td>Ifetch</td>
<td>Reg</td>
</tr>
<tr>
<td>Ifetch</td>
<td>Reg</td>
</tr>
<tr>
<td>R-type</td>
<td></td>
</tr>
</tbody>
</table>
Why Pipeline?

° Suppose we execute 100 instructions

° Single Cycle Machine
  • 45 ns/cycle x 1 CPI x 100 inst = 4500 ns

° Multicycle Machine
  • 10 ns/cycle x 4.1 CPI (due to inst mix) x 100 inst = 4100 ns

° Ideal pipelined machine
  • 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns
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
Why Pipeline? Because we can!

Time (clock cycles)

Inst 0
Inst 1
Inst 2
Inst 3
Inst 4
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)

• 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

• 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

° 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 \((1 \times .5 + 2 \times .5) = 1.5\)
  - 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

add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or r8,r1,r9
xor r10,r1,r11
Data Hazard on r1:

- Dependencies backwards in time are hazards

Instruction Order

<table>
<thead>
<tr>
<th>add r1, r2, r3</th>
</tr>
</thead>
<tbody>
<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:

• “Forward” result from one stage to another

```
<table>
<thead>
<tr>
<th>Instruction Order</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>
```

• “or” OK if define read/write properly
Forwarding (or Bypassing): What about Loads?

- Dependencies backwards in time are hazards

\[ \text{lw } r_1,0(r_2) \]
\[ \text{sub } r_4,r_1,r_3 \]

- Can’t solve with forwarding:
- Must delay/stall instruction dependent on loads
Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards

\[ \text{lw } r1,0(r2) \]
\[ \text{sub } r4,r1,r3 \]

- 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

° 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 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

Exec

Next PC

PC

Inst. Mem

Reg File

IR

A

B

S

Mem Access

M

Mem

Data Mem

Reg. File

D

3/1/04

©UCB Spring 2004

Lec10.31
Summary: Pipelining

° Reduce CPI by overlapping many instructions
  • Average throughput of approximately 1 CPI with fast clock

° Utilize capabilities of the Datapath
  • 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

° 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
Reconfigurable Computing

Pipelining II

Philip Leong (philip.leong@sydney.edu.au)
School of Electrical and Information Engineering
http://www.ee.usyd.edu.au/~phwl

lecture slides adapted from Patterson and Hennesy,
“Computer Organization and Design”: http://inst.eecs.berkeley.edu/~cs152/
Introduction to Pipelining: Datapath and Control

March 3rd, 2004

John Kubiatowicz (www.cs.berkeley.edu/~kubitron)

lecture slides: http://inst.eecs.berkeley.edu/~cs152/
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
Data Hazard on r1: Read after write hazard (RAW)

```plaintext
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)

\[
\text{lw } r1, 0(r2) \quad \text{sub } r4, r1, r3
\]

- Can't solve with forwarding:
- Must delay/stall instruction dependent on loads
Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards

```
Time (clock cycles)
```

```
lw r1,0(r2)
```

```
sub r4,r1,r3
```

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

Structural Hazard

Control Hazard

RAW (read after write) Data Hazard

WAW Data Hazard (write after write)

WAR Data Hazard (write after read)
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 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

Next PC
PC
Inst. Mem
Reg. File
Exec
S
Mem Access
M
Data Mem
Reg. File

Equal

A
B
D

3/3/04
©UCB Spring 2004
What happens if we start a new instruction every cycle?
Pipelined Datapath (as in book); hard to read
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

<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>Wr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 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
    
    ![Load Pipeline Diagram]
  
  • R-type uses Register File’s Write Port during its 4th stage
    
    ![R-type Pipeline Diagram]

° 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.

<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>
<th>Cycle 8</th>
<th>Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<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>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>Load</td>
<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>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>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>
</tbody>
</table>
Modified Control & Datapath

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] ← M;
R[rt] ← M;
R[rd] ← M;

Exec

Reg. File

Mem Access

Data Mem

IR

PC

Next PC

Inst. Mem

A

B

S

M

D

3/3/04
©UCB Spring 2004

CS152 / Kubiatowicz

Lec10.50
The Four Stages of Store

- **Cycle 1**: Ifetch
- **Cycle 2**: Reg/Dec
- **Cycle 3**: Exec
- **Cycle 4**: Mem

**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
IR <- Mem[PC]; PC < PC+4;

A <- R[rs]; B <– R[rt]

S ← A + B;
M ← S
R[rd] ← S;

S ← A or ZX;
M ← S
R[rt] ← S;

S ← A + SX;
M ← Mem[S]
R[rd] ← M;

S ← A + SX;
Mem[S] <- B

If Cond PC < PC+SX;

M <- S
Reg. File

Next PC
Inst. Mem
Reg File
Exec
Mem Access
Data Mem
M

IR
PC
A
B
D

Equal

3/3/04
©UCB Spring 2004

CS152 / Kubiatowicz
Lec10.53
Recall: Single cycle control!

Datapath

Control

Ideal Instruction Memory

Instruction Address

Next Address

PC

Rw Ra Rb
32 32-bit Registers

A
32

B

32

ALU

Conditions

Control Signals

Data Memory

Ideal Data Memory

Instruction

Rd Rs Rt
5 5 5

Data In

Data Out

Clk

Next Address

32

Clk

32

Clk
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
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

lw r1, r2(35)

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

- **IF (Instruction Fetch):**
  - Instruction Mem: `addl r2, r2, 3`
  - IR (Instruction Register): `lw r1`
  - Reg File: `lw r1`
  - PC (Program Counter): 20

- **ID (Instruction Decode):**
  - Reg File: `addl r2, r2, 3`
  - Execution: `lw r1`
  - Next PC: 20

- **EX (Execution):**
  - Exec: `lw r1, r2(35)`
  - Mem Access: `add r10, r11, r12`
  - Mem Ctrl: `ori r8, r9, 17`
  - WB Ctrl: `beq r6, r7, 100`
  - Reg File: `add r10, r11, r12`
  - Data Mem: `ori r8, r9, 17`
  - Mem Access: `and r13, r14, 15`

- **WB (Write Back):**
  - WB Ctrl: `lw r1, r2(35)`
  - Reg File: `add r10, r11, r12`
  - Mem Access: `ori r8, r9, 17`
  - Data Mem: `and r13, r14, 15`

3/3/04
Fetch 24, Decode 20, Exec 14, Mem 10

```
lw r1, r2(35)  # 10
add r2, r2, 3  # 14
sub r3, r4, r5 # 20
beq r6, r7, 100 # 24
ori r8, r9, 17 # 30
add r10, r11, r12 # 34
and r13, r14, 15 # 100
```
Note Delayed Branch: always execute ori after beq
Fetch 100, Dcd 30, Ex 24, Mem 20, WB 14

IR
Inst. Mem
ori r8, r9, 17

Reg File
r6
r7

Decide
beg

Exec
sub r3

Mem Ctrl
add r2

Reg File
r1=M[r2+35]

WB
PC

Next PC
= 100

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

©UCB Spring 2004
Fill it in yourself!
Fill it in yourself!
Pipelined Processor

- Separate control at each stage
- Stalls propagate backwards to freeze previous stages
- Bubbles in pipeline introduced by placing “Noops” into local stage, stall previous stages.
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)
Is CPI = 1 for our pipeline?

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

IFetch  Dcd  Exec  Mem  WB

IFetch  Dcd  Exec  Mem  WB

IFetch  Dcd  Exec  Mem  WB

IFetch  Dcd  Exec  Mem  WB

° 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)