CPS 104
Computer Organization and Programming
Lecture 24: A Pipelined Processor

Dietolf (Dee) Ramm
Nov. 22, 1999
http://www.cs.duke.edu/~dr/cps104.html
Outline of Today’s Lecture

- Pipelined Datapath and Pipelined Control
- Pipeline Example: Instructions Interaction
- Pipeline Hazards
  - Data hazards
  - delayed load
  - Branch hazards, delayed branch
- Summary

What you should know:
  - Basic concept of pipelining
  - How long will a sequence of instructions take to execute on a single cycle processor, a pipelined processor?
  - Some of the complications introduced by pipelining
Pipelining: Its 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

- Sequential laundry takes 6 hours for 4 loads.
- If they learned pipelining, how long would laundry take?

Task Order

- Time
  - 6 PM
  - 7
  - 8
  - 9
  - 10
  - 11
  - Midnight

- Time:
  - 30
  - 40
  - 20
  - 30
  - 40
  - 20
  - 30
  - 40
  - 20

A

B

C

D

- 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.

- Potential speedup = Number pipe stages.

- Unbalanced lengths of pipe stages reduces speedup.

- Time to “fill” pipeline and time to “drain” it reduces speedup.
A Multiple Cycle Implementation

- The root of the single cycle processor’s problems:
  - The cycle time has to be long enough for the slowest instruction

- Solution:
  - Break the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
    - Cycle time: time it takes to execute the longest step
    - Keep all the steps so they have similar length
  - This is the essence of the multiple cycle processor

- The advantages of the multiple cycle processor:
  - Cycle time is much shorter
  - Different instructions take different number of cycles to complete
    - Load takes five cycles
    - Jump only takes three cycles
  - Allows a functional unit to be used more than once per instruction
Register-Register Timing

- Instruction Memory Access Time
- Delay through Control Logic
- Register File Access Time
- ALU Delay

Register Write Occurs Here
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

- **WrB**: Write the data back to the register file
Key Ideas Behind Instruction Execution Pipelining

- Overlap execution of instructions
- The load instruction has 5 stages: I-fetch, Reg-Fetch / I-Decode, Execute, Memory-Access, Register Write-Back.
  - Five independent functional units to work on each stage
    - Each functional unit is used only once
  - The 2nd load can start as soon as the 1st finishes its I-fetch stage
  - Each load still takes five cycles to complete. latency is still 5 cycles
  - The throughput is much higher; CPI is 1 with ~1/5 cycle time.
  - Instructions start before the previous ones are completed.
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 WrB stage

- One instruction enters the pipeline every cycle
  - One instruction comes out of the pipeline (completed) every cycle
  - The “Effective” Cycles per Instruction (CPI) is 1; ~1/5 cycle time
The Four Stages of R-type

- **Ifetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Register access and Instruction Decode
- **Exec**: ALU operates on the two register operands
- **WrB**: Write the ALU output back to the register file
Pipelining the R-type and Load Instruction

We have a problem called pipeline conflict or resource hazard:

- Two instructions try to write to the register file at the same time!
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

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
\text{Load} & \text{Ifetch} & \text{Reg/Dec} & \text{Exec} & \text{Mem} & \text{WrB} \\
\hline \\
\text{R-type} & \text{Ifetch} & \text{Reg/Dec} & \text{Exec} & \text{WrB} \\
\end{array}
\]

How to solve this pipeline hazard?
**Solution: 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 **NO-OP** stage: nothing is being done. Effective CPI?

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

(R-type)
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 Four Stages of Beq

- **Ifetch:** Instruction Fetch
  - Fetch the instruction from the Instruction Memory

- **Reg/Dec:** Registers Fetch and Instruction Decode

- **Exec:** ALU compares the two register operands
  - Adder calculates the branch target address

- **Mem:** If the registers we compared in the Exec stage are the same,
  - Write the branch target address into the PC
A Pipelined Datapath
The Instruction Fetch Stage

- Location 10: lw $1, 0x100($2)  $1 <- Mem[($2) + 0x100]
A Detail View of the Instruction Fetch Unit

- Location 10: lw $1, 0x100($2)

![Diagram showing the instruction fetch unit pipeline stages: Clk, Ifetch, Adder, Reg/Dec, and the memory access with an address calculation and instruction retrieval process.](image-url)
The Decode / Register Fetch Stage

- Location 10: `lw $1, 0x100($2)`
  
  
  
  $1 \leftarrow \text{Mem}[(\$2) + 0x100]$

You are here!

- Pipeline stages: IF/ID, ID/Ex, Ex/Mem, Mem, etc.
Load’s Address Calculation Stage

- Location 10: \texttt{lw} $1, 0x100($2) \quad $1 \leftarrow \text{Mem}[(\text{R2}) + 0x100]

You are here!
Load’s Memory Access Stage

Location 10: `lw $1, 0x100($2)`  
$1 <- \text{Mem}[($2) + 0x100]$
Load’s Write Back Stage

- Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]

You are somewhere out there!
A More Extensive Pipelining Example

Clock

End of Cycle 4: Load’s Mem, R-type’s Exec, Store’s Reg, Beq’s Ifetch

End of Cycle 5: Load’s WrB, R-type’s Mem, Store’s Exec, Beq’s Reg

End of Cycle 6: R-type’s WrB, Store’s Mem, Beq’s Exec

End of Cycle 7: Store’s WrB, Beq’s Mem
Pipelining Example: End of Cycle 4

0: Load’s Mem  4: R-type’s Exec  8: Store’s Reg  12: Beq’s Ifetch
Pipelining Example: End of Cycle 5

- 0: Lw’s Wr  4: R’s Mem  8: Store’s Exec  12: Beq’s Reg  16: R’s Ifetch

**0: Load’s Wr**

**12: Beq’s Reg**

**8: Store’s Exec**

**4: R-type’s Mem**

**16: R’s Ifetch**

**0: Load’s Wr**
Pipelining Example: End of Cycle 6

- 4: R’s Wr
- 8: Store’s Mem
- 12: Beq’s Exec
- 16: R’s Reg
- 20: R’s Ifet

IF/ID: Instruction @ 20

ID/Ex: R-type’s busA & B

Ex/Mem: Beq’s Results

Mem/Wr: Nothing for St

PC = 24

RegWr = 1

ALUOp = Sub

ExtOp = 1

Branch = 0

ExtOp = 1

MemWr = 1

MemtoReg = 0

R-type’s Ifet

R-type’s Reg

R-type’s Reg

Store’s Mem

R-type’s Wr
Pipelining Example: End of Cycle 7

- 8: Store’s Wr  12: Beq’s Mem  16: R’s Exec  20: R’s Reg  24: R’s Ifet
Data Hazards

- So far we ignored instructions dependencies, but in a real machine one must deal with dependencies.

- Example:
  
  \[
  \begin{align*}
  &\text{sub} \quad \$2, \quad \$1, \quad \$3 \\
  &\text{and} \quad \$12, \quad \$2, \quad \$5 \quad \# \quad \$12 \text{ depends on the result in } \$2 \\
  &\text{or} \quad \$13, \quad \$6, \quad \$2 \quad \# \quad \text{but } \$2 \text{ is updated 3 clock} \\
  &\text{add} \quad \$14, \quad \$2, \quad \$2 \quad \# \quad \text{cycles later.} \\
  &\text{sw} \quad \$15, \quad 100(\$2) \quad \# \quad \text{We have a problem!!}
  \end{align*}
  \]
Data Hazard Solution: Register Forwarding

![Diagram of pipeline stages with forward unit](image_url)
The Delay Branch Phenomenon

- Although Beq is fetched during Cycle 4:
  - Target address is **NOT** written into the PC until the end of Cycle 7
  - Branch’s target is **NOT** fetched until Cycle 8
  - 3-instruction delay before the branch take effect

- This is referred to as **Branch Hazard**:
  - Clever design techniques can reduce the delay to **ONE instruction**
Reducing Branch delays (cont.)

- The design is optimized for “branch not taken” (no pipeline delay)
- If branch is taken, the next instruction is converted to NOOP by the control (“pipeline bubble” <=> one stage pipeline delay).
- The MIPS architecture defines a delayed Branch slot to reduce this potential delay (see a later slide).
The Delay Load Phenomenon

- Although Load is fetched during Cycle 1:
  - The data is NOT written into the Reg File until the end of Cycle 5
  - We cannot read this value from the Reg File until Cycle 6
  - 3-instruction delay before the load take effect

- This is referred to as Data Hazard:
  - Register forwarding reduces the load delay to ONE instruction
  - It is not possible to entirely eliminate the load delay.
Delayed Load and Branch on a Real MIPS Processor

- The effect of load in a real MIPS Processor is delayed:
  - \( \text{lw} \ $1, 100 \ ($2) \) // Load Register R1
  - \( \text{add} \ $3, $1, $0 \) // Move “old” R1 into R3
  - \( \text{add} \ $4, $1, $0 \) // Move “new” R1 into R4
- The effect of load on a “normal processor” is NOT delayed:
  - \( \text{lw} \ $1, 100 \ ($2) \) // Load Register R1
  - \( \text{add} \ $3, $1, $0 \) // Move “new” R1 into R3

- The effect of branch and jump in a real MIPS Processor is delayed:
  - Instruction Address: 0x00 \( j \ 1000 \)
  - Instruction Address: 0x04 \( \text{add} \ $1, $2, $3 \)
  - Instruction Address: 0x1000 \( \text{sub} \ $1, $2, $3 \)
- Branch and jump in a “Normal processor” are NOT delayed:
  - Instruction Address: 0x00 \( j \ 1000 \)
  - Instruction Address: 0x1000 \( \text{sub} \ $1, $2, $3 \)
CPU design Summary

- Disadvantages of the **Single Cycle Processor**
  - Long cycle time
  - Cycle time is too long for all instructions except the Load

- **Multiple Clock Cycle Processor**:
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle

- **Pipeline Processor**:
  - Natural enhancement of the multiple clock cycle processor
  - Each functional unit can only be used once per instruction
  - If a instruction is going to use a functional unit:
    - it must use it at the same stage as all other instructions

- **Pipeline Control**:
  - Each stage’s control signal depends ONLY on the instruction that is currently in that stage
Additional Notes

- All Modern CPUs use pipelines.
- Many CPUs have 8-12 pipeline stages.
- The latest generation processors (Pentium-II, PowerPC-604 or G3, DEC Alpha 21164, SUN’s UltraSPARC) use multiple pipelines to get higher speed (Superscalar design).
- The course: CPS220: Advanced Computer Architecture I covers the design of Pipelined and Superscalar processors.