CS429: Computer Organization and Architecture
Pipeline I

Warren Hunt, Jr. and Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: November 3, 2014 at 07:54
Overview

What’s wrong with the sequential (SEQ) Y86?

- It’s slow!
- Each piece of hardware is used only a small fraction of the time.
- We would like to find a way to get more performance with only a little more hardware.

General Principles of Pipelining

- Express task as a collection of stages
- Move instructions through stages
- Process several instructions at any given moment

Creating a Pipelined Y86 Processor

- Rearrange SEQ
- Insert pipeline registers
- Deal with data and control hazards
Suppose you have four folks, each with a load of clothes to wash, dry, fold and stash away. There are four subtasks: wash, dry, fold, stash. Each takes 30 minutes.

Time to do a load of laundry from start to finish: 2 hours.
Sequential laundry takes 8 hours for 4 loads.
If they learned pipelining, how long would laundry take?
Pipelined laundry takes 3.5 hours for 4 loads!
Pipelining Lessons

- Pipelining doesn’t help *latency* of a single task; it helps *throughput* of the entire workload.
- Multiple tasks operate simultaneously using different resources.
- Potential speedup = number of stages.
- Unbalanced lengths of pipe stages reduces speedup.
- Time to “fill” pipeline and time to “drain” it reduces speedup.
- Stall for dependencies.
System
- Computation requires a total of 300 picoseconds.
- Needs an additional 20 picoseconds to save the result in the register.
- Must have a clock cycle of at least 320 ps. Why?
System

- Divide combinational logic into 3 blocks of 100 ps each.
- Can begin a new operation as soon as the previous one passes through stage A.
- Begin new operation every 120 ps. Why?
- Overall latency *increases*! It’s now 360 ps from start to finish.
Pipeline Diagrams

**Unpipelined**

Cannot start new operation until the previous one completes.

**3-Way Pipelined**

Up to 3 operations in process simultaneously.
At times [240..260].
Limitations: Non-uniform Delays

Throughput is limited by the slowest stage.
Other stages may sit idle for much of the time.
It’s challenging to partition the system into balanced stages.
As you try to deepen the pipeline, the overhead of loading registers becomes more significant.

**Percentage of clock cycle spend loading registers:**

<table>
<thead>
<tr>
<th>Pipeline Type</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-stage pipeline:</td>
<td>6.25%</td>
</tr>
<tr>
<td>3-stage pipeline:</td>
<td>16.67%</td>
</tr>
<tr>
<td>6-stage pipeline:</td>
<td>28.57%</td>
</tr>
</tbody>
</table>

High speeds of modern processor designs are obtained through very deep pipelining.
The Performance Equation

\[
\text{CPU Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}
\]

Clock Cycle Time
- Improves by a factor of almost N for N-deep pipeline.
- Not quite a factor of N due to pipeline overheads.

Cycles Per Instructions
- In an ideal world, CPI would stay the same.
- An individual instruction takes N cycles.
- But we have N instructions in flight at a time.
- So, average \( CPI_{pipe} = \frac{CPI_{no-pipe} \times N}{N} \)

Thus, performance can improve by up to a factor of N.
Sequential System: Each operation depends on the previous one.
Pipelined System:
- Result does not feed back around in time for the next operation.
- Pipelining has changed the behavior of the system.
Data Dependencies in Processors

Result from one instruction is used as an operand for another; called read-after-write (RAW) dependency.

- This is very common in actual programs.
- Must make sure that our pipeline handles these properly and gets the right result.
- Should minimize performance impact as much as possible.

irmovl $50, %eax
mrmovl 100( %ebx ), %edx
addl %eax, %ebx

This is very common in actual programs.
Must make sure that our pipeline handles these properly and gets the right result.
Should minimize performance impact as much as possible.
Stages occur in sequence.

One operation in process at a time.

One stage for each logical pipeline operation.

- **Fetch**: get next instruction from memory.
- **Decode**: figure out what to do, and get values from regfile.
- **Execute**: compute.
- **Memory**: access data memory if needed.
- **Write back**: write results to regfile, if needed.
Still sequential implementation, but reorder PC stage to put at the beginning

**PC Stage**
- Task is to select PC for current instruction.
- Based on results computed by previous instruction.

**Processor State**
- PC is no longer stored in a register.
- But, can determine PC based on other stored information.