

Lecture #22 CPU Design: Pipelining to Improve Performance II

2007-8-1



Scott Beamer, Instructor



Review: Processor Pipelining (1/2)

- “Pipeline registers” are added to the datapath/controller to neatly divide the single cycle processor into “pipeline stages”.
- **Optimal Pipeline**
 - Each stage is executing part of an instruction each clock cycle.
 - One inst. finishes during each clock cycle.
 - On average, execute far more quickly.
- **What makes this work well?**
 - Similarities between instructions allow us to use same stages for all instructions (generally).
 - Each stage takes about the same amount of time as all others: little wasted time.



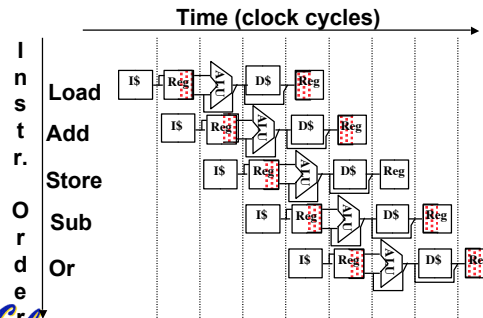
Review: Pipeline (2/2)

- **Pipelining is a BIG IDEA**
 - widely used concept
- **What makes it less than perfect?**
 - **Structural hazards:** Conflicts for resources. Suppose we had only one cache? ⇒ Need more HW resources
 - **Control hazards:** Branch instructions effect which instructions come next. ⇒ Delayed branch
 - **Data hazards:** Data flow between instructions. ⇒ Forwarding

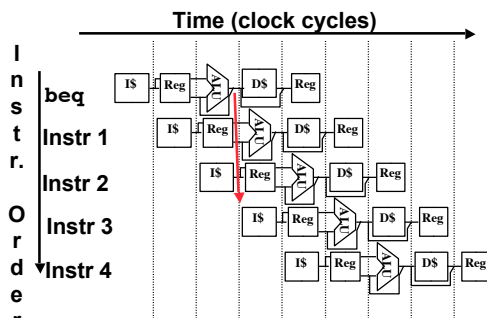


Graphical Pipeline Representation

(In Reg, right half highlight read, left half write)



Control Hazard: Branching (1/8)



Where do we do the compare for the branch?



Control Hazard: Branching (2/8)

- We had put branch decision-making hardware in ALU stage
 - therefore two more instructions after the branch will *always* be fetched, whether or not the branch is taken
- **Desired functionality of a branch**
 - if we do not take the branch, don't waste any time and continue executing normally
 - if we take the branch, don't execute any instructions after the branch, just go to the desired label



Control Hazard: Branching (3/8)

- Initial Solution: Stall until decision is made
- insert “no-op” instructions (those that accomplish nothing, just take time) or hold up the fetch of the next instruction (for 2 cycles).
- Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)

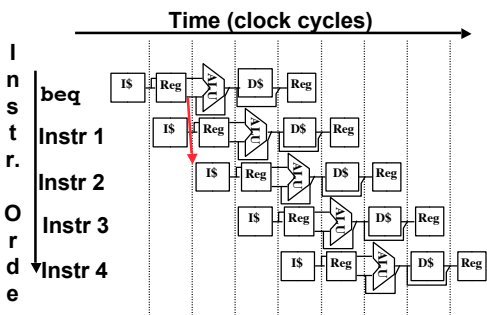


Control Hazard: Branching (4/8)

- Optimization #1:
 - insert **special branch comparator** in Stage 2
 - as soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
 - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
 - Side Note: This means that branches are idle in Stages 3, 4 and 5.



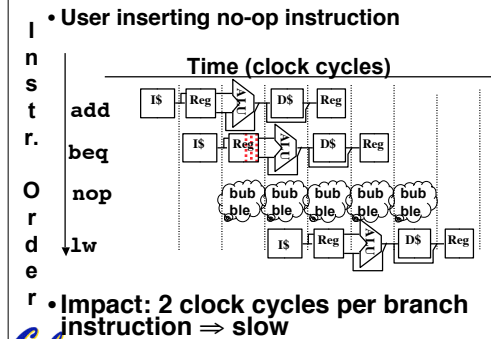
Control Hazard: Branching (5/8)



Branch comparator moved to Decode stage.



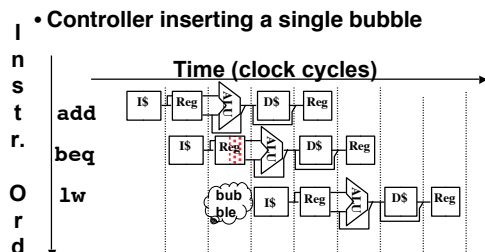
Control Hazard: Branching (6a/8)



Impact: 2 clock cycles per branch instruction ⇒ slow



Control Hazard: Branching (6b/8)



Impact: 2 clock cycles per branch instruction ⇒ slow



Control Hazard: Branching (7/8)

- Optimization #2: Redefine branches
 - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
 - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (called the **branch-delay slot**)
 - The term “**Delayed Branch**” means we **always** execute inst after branch
 - **This optimization is used on the MIPS**



Control Hazard: Branching (8/8)

Notes on Branch-Delay Slot

- **Worst-Case Scenario:** can always put a no-op in the branch-delay slot
- **Better Case:** can find an instruction preceding the branch which can be placed in the branch-delay slot without affecting the flow of the program
 - re-ordering instructions is a common method of speeding up programs
 - compiler must be very smart in order to find instructions to do this
 - usually can find such an instruction at least 50% of the time
 - Jumps also have a delay slot...



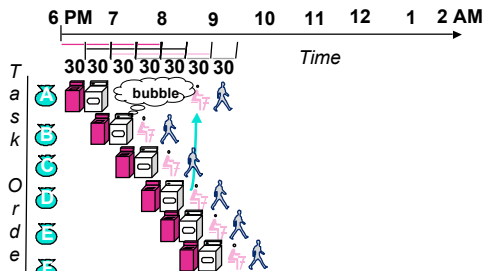
Example: Nondelayed vs. Delayed Branch

Nondelayed Branch	Delayed Branch
or \$8, \$9, \$10	add \$1, \$2, \$3
add \$1, \$2, \$3	sub \$4, \$5, \$6
sub \$4, \$5, \$6	beq \$1, \$4, Exit
beq \$1, \$4, Exit	or \$8, \$9, \$10
xor \$10, \$1, \$11	xor \$10, \$1, \$11

Exit:



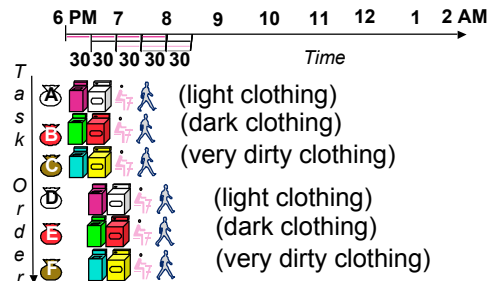
Out-of-Order Laundry: Don't Wait



A depends on D; rest continue; need more resources to allow out-of-order



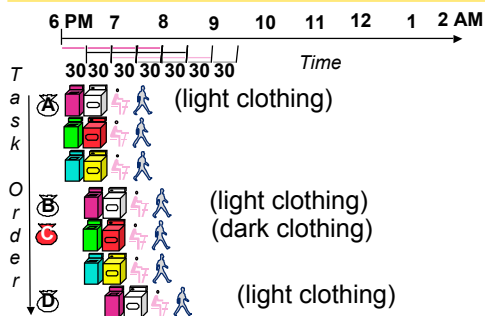
Superscalar Laundry: Parallel per stage



More resources, HW to match mix of parallel tasks?



Superscalar Laundry: Mismatch Mix

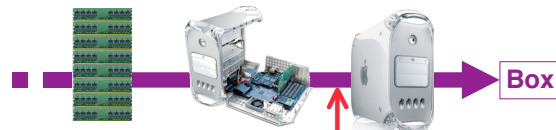


Task mix underutilizes extra resources



Real-world pipelining problem

• You're the manager of a HUGE assembly plant to build computers.



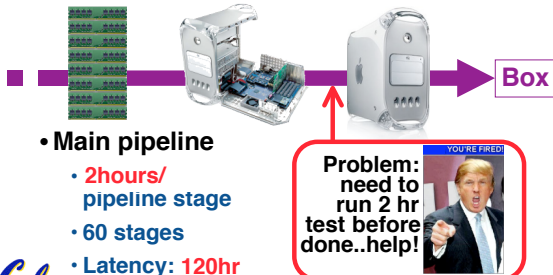
- Main pipeline
 - 10 minutes/pipeline stage
 - 60 stages
 - Latency: 10hr

Problem: need to run 2 hr test before done..help!



Real-world pipelining problem solution 1

- You remember: “a pipeline frequency is limited by its slowest stage”, so...



- Main pipeline
 - 2hours/ pipeline stage
 - 60 stages
 - Latency: 120hr

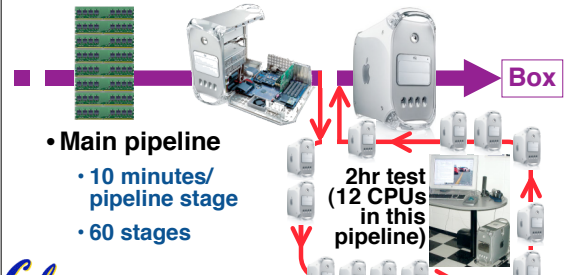


CS61C L22 CPU Design : Pipelining to Improve Performance II (19)

Beamer, Summer 2007 © UCB

Real-world pipelining problem solution 2

- Create a sub-pipeline!



- Main pipeline
 - 10 minutes/ pipeline stage
 - 60 stages



CS61C L22 CPU Design : Pipelining to Improve Performance II (2)

Beamer, Summer 2007 © UCB

Peer Instruction (1/2)

Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after 10^3 loops, so pipeline full)

```
Loop:   lw    $t0, 0($s1)
        addu $t0, $t0, $s2
        sw    $t0, 0($s1)
        addiu $s1, $s1, -4
        bne  $s1, $zero, Loop
        nop
```

- How many pipeline stages (clock cycles) per loop iteration to execute this code?

1
2
3
4
5
6
7
8
9
10

© 2007 © UCB

Peer Instruction (2/2)

Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after 10^3 loops, so pipeline full). Rewrite this code to reduce pipeline stages (clock cycles) per loop to as few as possible.

```
Loop:   lw    $t0, 0($s1)
        addu $t0, $t0, $s2
        sw    $t0, 0($s1)
        addiu $s1, $s1, -4
        bne  $s1, $zero, Loop
        nop
```

- How many pipeline stages (clock cycles) per loop iteration to execute this code?

1
2
3
4
5
6
7
8
9
10

© 2007 © UCB

“And in Early Conclusion..”

- Pipeline challenge is hazards
 - Forwarding helps w/many data hazards
 - Delayed branch helps with control hazard in 5 stage pipeline
 - Load delay slot / interlock necessary
- More aggressive performance:
 - Superscalar
 - Out-of-order execution



CS61C L22 CPU Design : Pipelining to Improve Performance II (25)

Beamer, Summer 2007 © UCB

Administrivia

- Assignments
 - HW7 due 8/2
 - Proj3 due 8/5
- Midterm Regrades due Today
- Logisim in lab is now 2.1.6
 - `java -jar ~cs61c/bin/logisim`
- Valerie’s OH on Thursday moved to 10-11 for this week



CS61C L22 CPU Design : Pipelining to Improve Performance II (26)

Beamer, Summer 2007 © UCB

Why Doesn't It Work?

- **DO NOT MESS WITH THE CLOCK**
 - Crafty veterans may do it very rarely and carefully
- Doing so will cause unpredictable and hard to track **errors**
- Following slides are from CS 150 Lecture by Prof. Katz

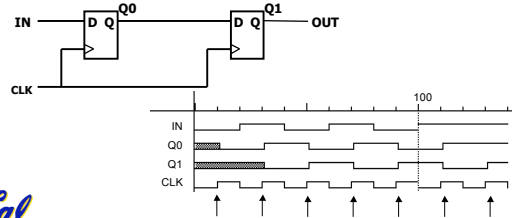


CS61C L22 CPU Design: Pipelining to Improve Performance II (27)

Beamer, Summer 2007 © UCB

Cascading Edge-triggered Flip-Flops

- **Shift register**
 - New value goes into first stage
 - While previous value of first stage goes into second stage
 - Consider setup/hold/propagation delays (prop must be > hold)

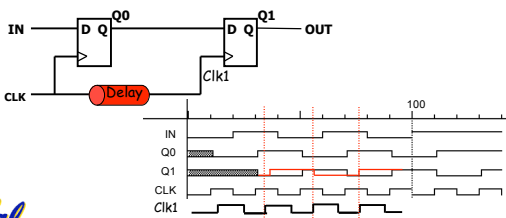


CS61C L22 CPU Design: Pipelining to Improve Performance II (28)

Beamer, Summer 2007 © UCB

Cascading Edge-triggered Flip-Flops

- **Shift register**
 - New value goes into first stage
 - While previous value of first stage goes into second stage
 - Consider setup/hold/propagation delays (prop must be > hold)

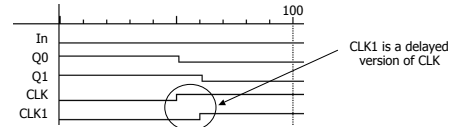


CS61C L22 CPU Design: Pipelining to Improve Performance II (29)

Beamer, Summer 2007 © UCB

Clock Skew

- **The problem**
 - Correct behavior assumes next state of all storage elements determined by all storage elements at the same time
 - Difficult in high-performance systems because time for clock to arrive at flip-flop is comparable to delays through logic (and will soon become greater than logic delay)
 - Effect of skew on cascaded flip-flops:



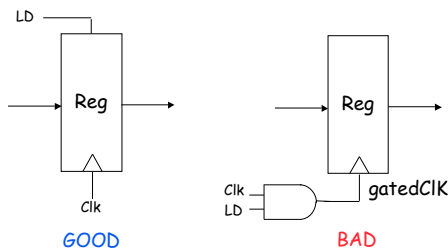
original state: IN = 0, Q0 = 1, Q1 = 1
due to skew, next state becomes: Q0 = 0, Q1 = 0, and not Q0 = 0, Q1 = 1



CS61C L22 CPU Design: Pipelining to Improve Performance II (30)

Beamer, Summer 2007 © UCB

Why Gating of Clocks is Bad!



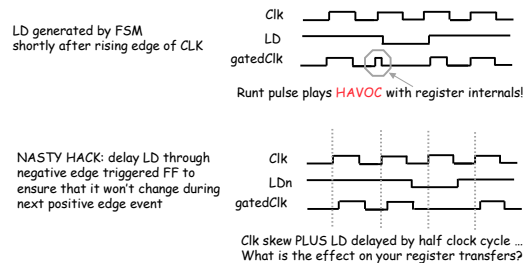
Do NOT Mess With Clock Signals!



CS61C L22 CPU Design: Pipelining to Improve Performance II (31)

Beamer, Summer 2007 © UCB

Why Gating of Clocks is Bad!



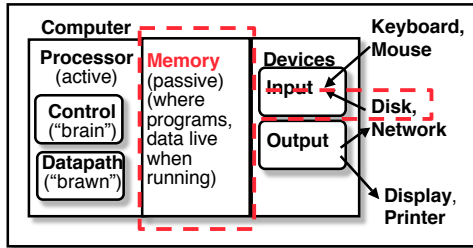
Do NOT Mess With Clock Signals!



CS61C L22 CPU Design: Pipelining to Improve Performance II (32)

Beamer, Summer 2007 © UCB

The Big Picture



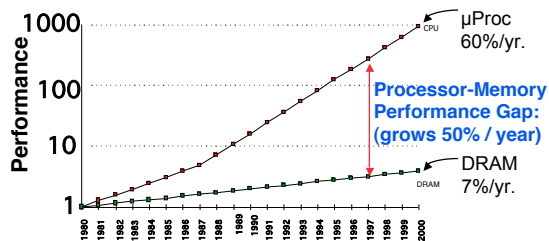
Memory Hierarchy

Storage in computer systems:

- **Processor**
 - holds data in register file (~100 Bytes)
 - Registers accessed on nanosecond timescale
- **Memory (we'll call "main memory")**
 - More capacity than registers (~Gbytes)
 - Access time ~50-100 ns
 - Hundreds of clock cycles per memory access?!
- **Disk**
 - HUGE capacity (virtually limitless)
 - VERY slow: runs ~milliseconds



Motivation: Why We Use Caches (written \$)



- 1989 first Intel CPU with cache on chip
- 1998 Pentium III has two levels of cache on chip

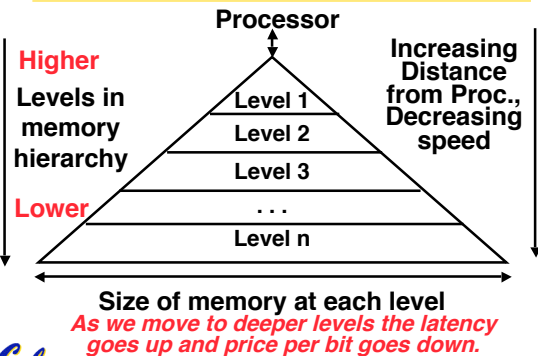


Memory Caching

- Mismatch between processor and memory speeds leads us to add a new level: a memory **cache**
- Implemented with same IC processing technology as the CPU (usually integrated on same chip): faster but more expensive than DRAM memory.
- Cache is a copy of a subset of main memory.
- Most processors have separate caches for instructions and data.



Memory Hierarchy



Memory Hierarchy

- If level closer to Processor, it is:
 - smaller
 - faster
 - subset of lower levels (contains most recently used data)
- Lowest Level (usually disk) contains all available data (or does it go beyond the disk?)
- Memory Hierarchy presents the processor with the illusion of a very large very fast memory.

