

**Lecture 31 –
 Pipelined Execution, part II**

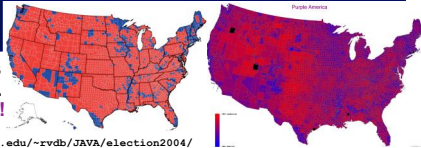
2004-11-10

Lecturer PSOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

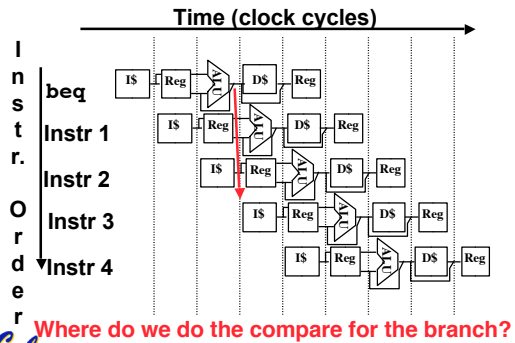


Election Data is now available...
 Purple Americal



www.princeton.edu/~rvdb/JAVA/election2004/
www.usatoday.com/news/politics/elections/vote2004/countymap.htm
 CS61C L31 Pipelined Execution, part II (1) Garcia, Fall 2004 © UCB

Control Hazard: Branching (1/7)



CS61C L31 Pipelined Execution, part II (4) Garcia, Fall 2004 © UCB

Control Hazard: Branching (2/7)

- We 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

CS61C L31 Pipelined Execution, part II (5) Garcia, Fall 2004 © UCB

Control Hazard: Branching (3/7)

- Initial Solution: Stall until decision is made
 - insert “no-op” instructions: those that accomplish nothing, just take time
 - Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)

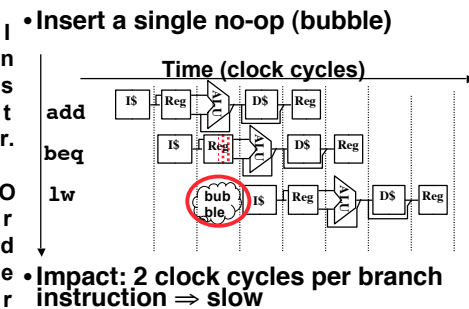
CS61C L31 Pipelined Execution, part II (6) Garcia, Fall 2004 © UCB

Control Hazard: Branching (4/7)

- Optimization #1:
 - move asynchronous comparator up to Stage 2
 - as soon as instruction is decoded (Opcode identifies is as a branch), immediately make a decision and set the value of the PC (if necessary)
 - 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.

CS61C L31 Pipelined Execution, part II (7) Garcia, Fall 2004 © UCB

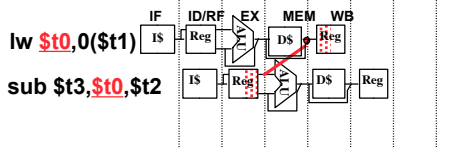
Control Hazard: Branching (5/7)



CS61C L31 Pipelined Execution, part II (8) Garcia, Fall 2004 © UCB

Data Hazard: Loads (1/4)

- Dependencies backwards in time are hazards



- Can't solve with forwarding
- Must stall instruction dependent on load, then forward (more hardware)

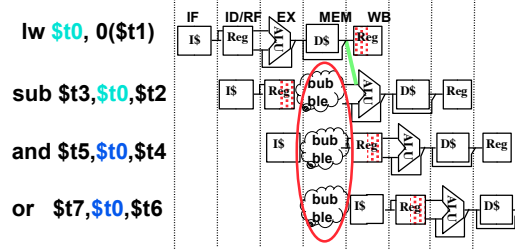


CS61C L31 Pipelined Execution, part II (19)

Garcia, Fall 2004 © UCB

Data Hazard: Loads (2/4)

- Hardware must stall pipeline
- Called “interlock”



CS61C L31 Pipelined Execution, part II (19)

Garcia, Fall 2004 © UCB

Data Hazard: Loads (3/4)

- Instruction slot after a load is called “load delay slot”
- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
- If the compiler puts an unrelated instruction in that slot, then no stall
- Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)

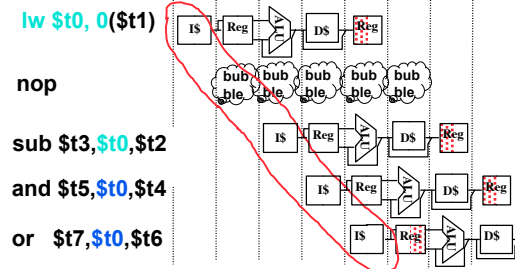


CS61C L31 Pipelined Execution, part II (17)

Garcia, Fall 2004 © UCB

Data Hazard: Loads (4/4)

- Stall is equivalent to nop



CS61C L31 Pipelined Execution, part II (18)

Garcia, Fall 2004 © UCB

Historical Trivia

- First MIPS design did not interlock and stall on load-use data hazard
- Real reason for name behind MIPS: **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
 - Word Play on acronym for Millions of Instructions Per Second, also called MIPS



CS61C L31 Pipelined Execution, part II (19)

Garcia, Fall 2004 © UCB

Administrivia

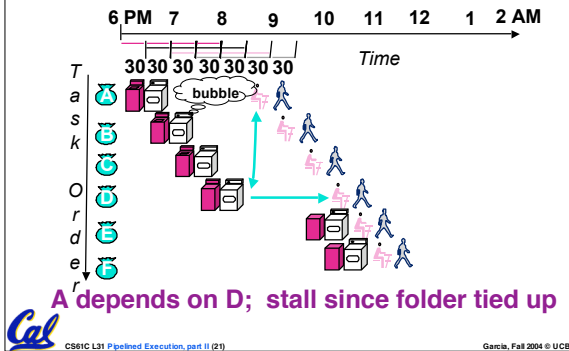
- No lab this week (wed, thu or fri)
 - Due to Veterans Day holiday on Thursday.
 - The lab is posted as a take-home lab; show TA your results in the following lab.
- Grade freezing update : through HW4
 - You have until next Wed to request regrades on HW3, HW4 & P1
- Back to 61C...Advanced Pipelining!
 - “Out-of-order” Execution
 - “Superscalar” Execution



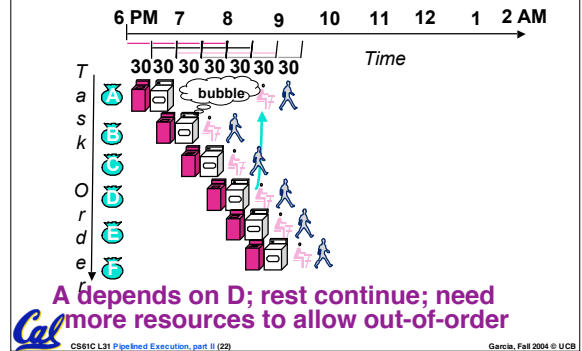
CS61C L31 Pipelined Execution, part II (20)

Garcia, Fall 2004 © UCB

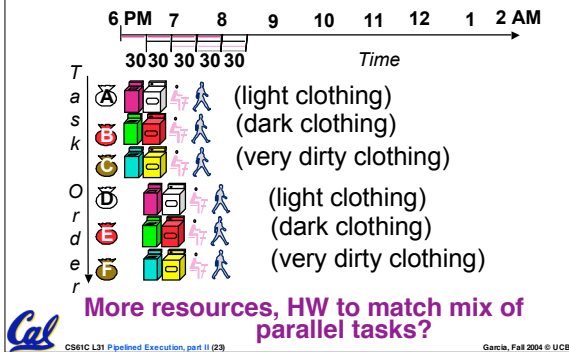
Review Pipeline Hazard: Stall is dependency



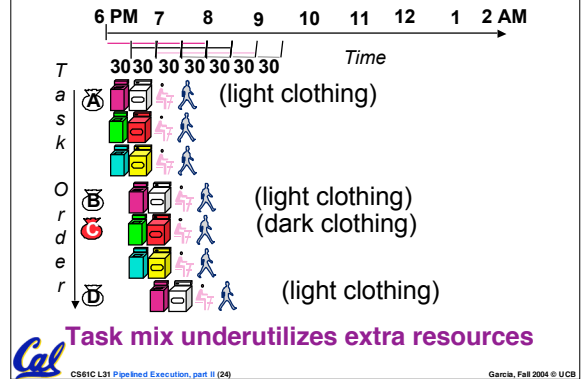
Out-of-Order Laundry: Don't Wait



Superscalar Laundry: Parallel per stage



Superscalar Laundry: Mismatch Mix



Peer Instruction

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

“And in Conclusion..”

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

