inst.eecs.berkeley.edu/~cs61c

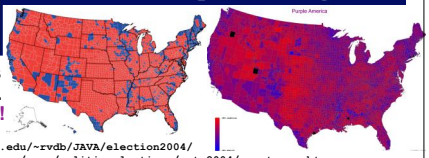
CS61C : Machine Structures

Lecture 31 – Pipelined Execution, part II

2004-11-10

Lecturer PSOE Dan Garcia

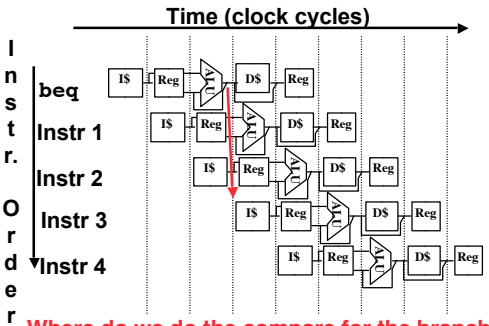
www.cs.berkeley.edu/~ddgarcia



Election Data is now available… Purple America!

www.princeton.edu/~rvdb/JAVA/election2004/
www.usatoday.com/news/politicselections/vote2004/countymap.htm

Control Hazard: Branching (1/7)



Where do we do the compare for the branch?

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

Control Hazard: Branching (3/7)

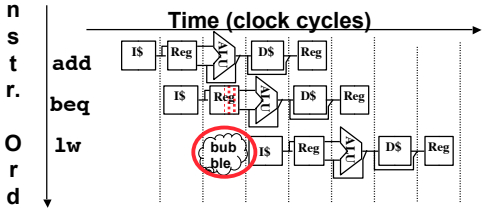
- Initial Solution: Stall until decision is made
  - insert "no-op" instructions: those that accomplish nothing, just take time
  - Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)

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.

Control Hazard: Branching (5/7)

- Insert a single no-op (bubble)



- Impact: 2 clock cycles per branch instruction ⇒ slow

Control Hazard: Branching (6/7)

- 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

Control Hazard: Branching (7/7)

- 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 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:	Exit:

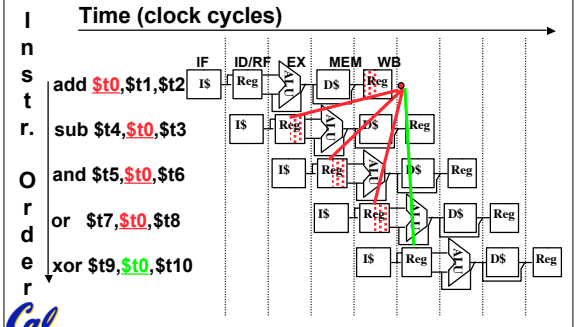
Data Hazards (1/2)

- Consider the following sequence of instructions

```
add $t0, $t1, $t2
sub $t4, $t0 ,$t3
and $t5, $t0 ,$t6
or  $t7, $t0 ,$t8
xor $t9, $t0 ,$t10
```

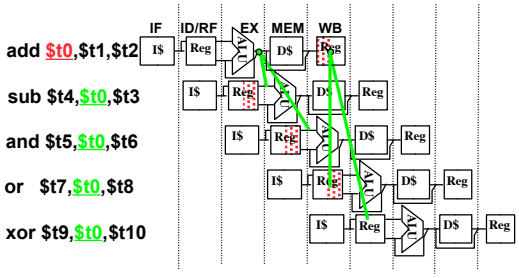

Data Hazards (2/2)

Dependencies backwards in time are hazards



Data Hazard Solution: Forwarding

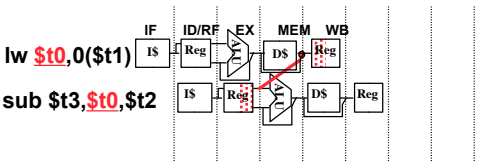
- Forward result from one stage to another



"or" hazard solved by register hardware

Data Hazard: Loads (1/4)

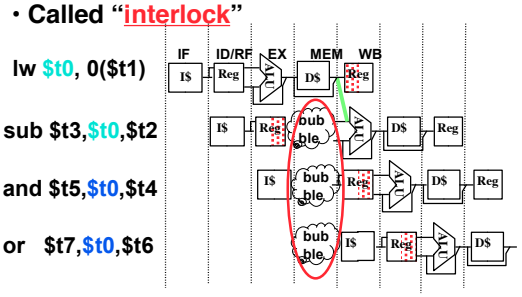
- Dependencies backwards in time are hazards



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

Data Hazard: Loads (2/4)

- Hardware must stall pipeline
- Called "interlock"

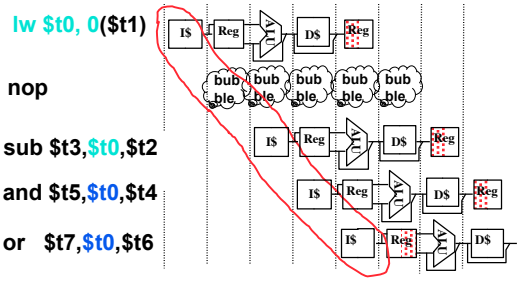


Data Hazard: Loads (3/4)

- 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)

Data Hazard: Loads (4/4)

- Stall is equivalent to nop



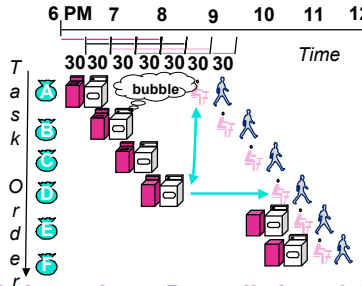
Historical Trivia

- First MIPS design did not interlock and stall on load-use data hazard
- Real reason for name behind MIPS:
  Microprocessor without
  Interlocked
  Pipeline
  Stages
  - Word Play on acronym for Millions of Instructions Per Second, also called MIPS

Administrivia

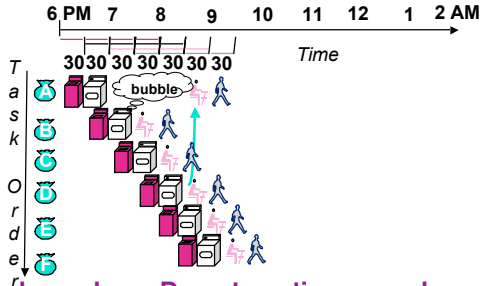
- No lab this week (wed, thu or fri)
  - Due to Veterans Day holiday on Thursday.
  - The lab is posted as a take-home lab; show TA your results in the following lab.
- Grade freezing update : through HW4
  - You have until next Wed to request regrades on HW3,HW4 & P1
- Back to 61C…Advanced Pipelining!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

Review Pipeline Hazard: Stall is dependency



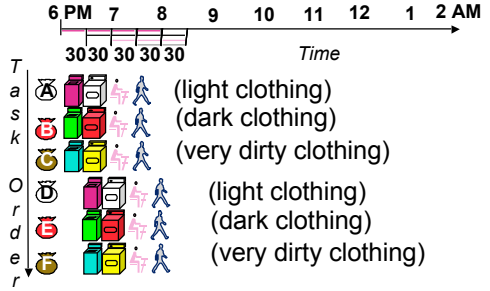
A depends on D; stall since folder tied up

Out-of-Order Laundry: Don't Wait



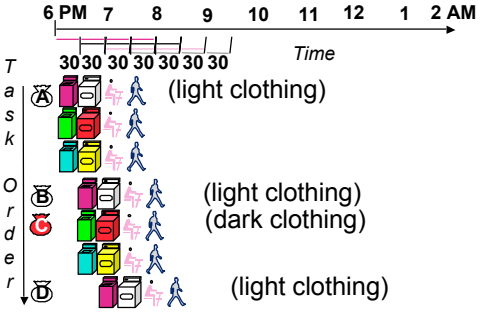
A depends on D; rest continue; need more resources to allow out-of-order

Superscalar Laundry: Parallel per stage



More resources, HW to match mix of parallel tasks?

Superscalar Laundry: Mismatch Mix



Task mix underutilizes extra resources

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