

CS61C - Machine Structures

Lecture 24 - Review Pipelined Execution

November 29, 2000

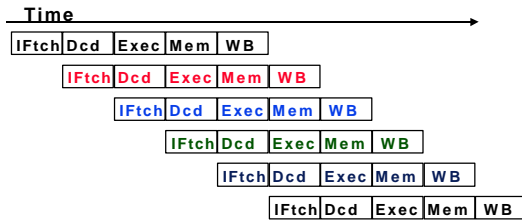
David Patterson

<http://www-inst.eecs.berkeley.edu/~cs61c/>

CS61C L24 Review Pipeline © UC Regents

1

Pipelined Execution Representation



° Every instruction must take same number of steps, also called pipeline “**stages**”, so some will go idle sometimes

CS61C L24 Review Pipeline © UC Regents

3

Problems for Computers

° Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle

- **Structural hazards:** HW cannot support this combination of instructions (e.g., read instruction and data from memory)
- **Control hazards:** Pipelining of branches & other instructions **stall** the pipeline until the hazard “**bubbles**” in the pipeline
- **Data hazards:** Instruction depends on result of prior instruction still in the pipeline (read and write same data)

CS61C L24 Review Pipeline © UC Regents

5

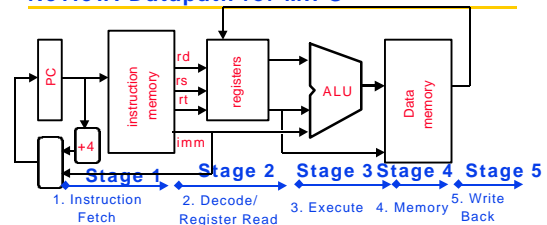
Steps in Executing MIPS

- 1) **IFetch:** Fetch Instruction, Increment PC
- 2) **Decode:** Instruction, Read Registers
- 3) **Execute:** Mem-ref: Calculate Address
Arith-log: Perform Operation
- 4) **Memory:** Load: Read Data from Memory
Store: Write Data to Memory
- 5) **Write Back:** Write Data to Register

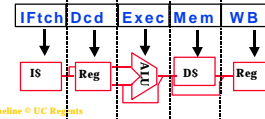
CS61C L24 Review Pipeline © UC Regents

2

Review: Datapath for MIPS



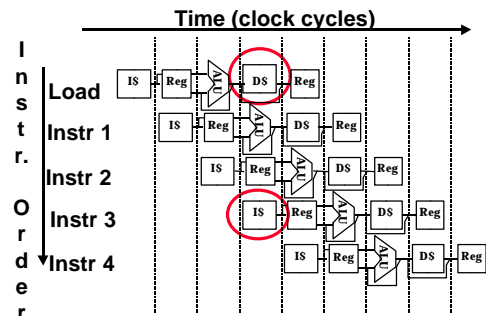
° Use datapath figure to represent pipeline



CS61C L24 Review Pipeline © UC Regents

4

Structural Hazard #1: Single Memory (1/2)



Read same memory twice in same clock cycle

CS61C L24 Review Pipeline © UC Regents

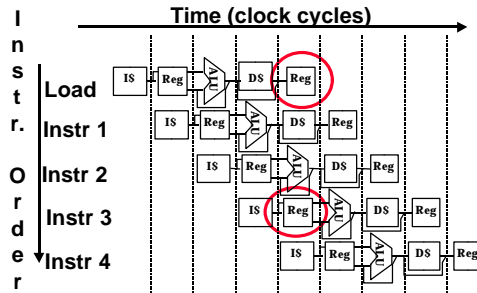
6

Structural Hazard #1: Single Memory (2/2)

◦ Solution:

- infeasible and inefficient to create second main memory
- so simulate this by having **two Level 1 Caches**
- have both an L1 **Instruction Cache** and an L1 **Data Cache**
- need more complex hardware to control when both caches miss

Structural Hazard #2: Registers (1/2)



Read and write registers simultaneously?

Structural Hazard #2: Registers (2/2)

◦ Solution:

- Build registers with multiple ports, so can both read and write at the same time
- What if read and write same register?
 - Design to that it writes in first half of clock cycle, read in second half of clock cycle
 - Thus will read what is written, reading the new contents

Data Hazards (1/2)

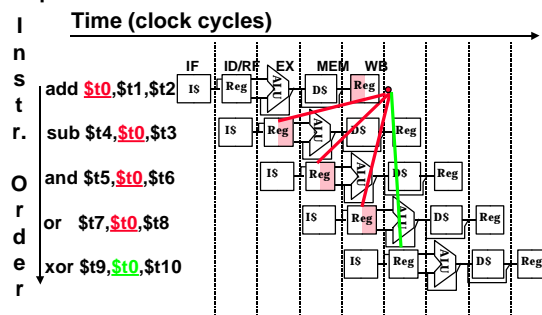
◦ Consider the following sequence of instructions

```

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

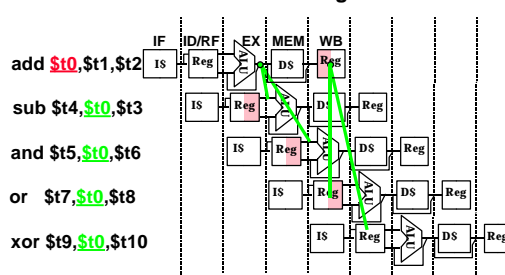
Data Hazards (2/2)

Dependencies backwards in time are hazards



Data Hazard Solution: Forwarding

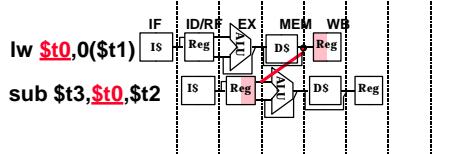
• Forward result from one stage to another



“or” hazard solved by register hardware

Data Hazard: Loads (1/2)

- Dependencies backwards in time are hazards



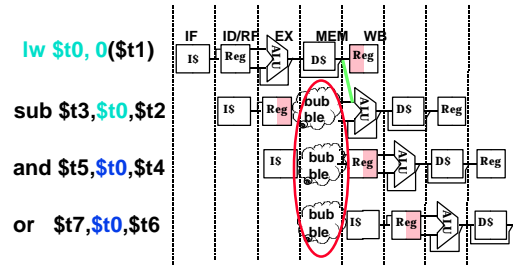
- Can't solve with forwarding
- Must stall instruction dependent on load, then forward (more hardware)

CS61C L24 Review Pipeline © UC Regents

13

Data Hazard: Loads (2/2)

- Hardware must insert no-op in pipeline



or \$t7, \$t0, \$t6

CS61C L24 Review Pipeline © UC Regents

14

Administrivia: Rest of 61C

- Rest of 61C slower pace

F 12/1 Review: Caches/TLB/VM; Section 7.5

M 12/4 Deadline to correct your grade record

W 12/6 Review: Interrupts (A.7); Feedback lab

F 12/8 61C Summary / Your Cal heritage / HKN Course Evaluation

Sun 12/10 Final Review, 2PM (155 Dwinelle)

Tues 12/12 Final (5PM 1 Pimintel)

◦ Final: Just bring pencils: leave home back packs, cell phones, calculators

◦ Will check that notes are handwritten

◦ Got a final conflict? Email now for Beta

CS61C

15

Control Hazard: Branching (1/6)

- Suppose we put branch decision-making hardware in ALU stage

- then 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 L24 Review Pipeline © UC Regents

16

Control Hazard: Branching (2/6)

- 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 L24 Review Pipeline © UC Regents

17

Control Hazard: Branching (3/6)

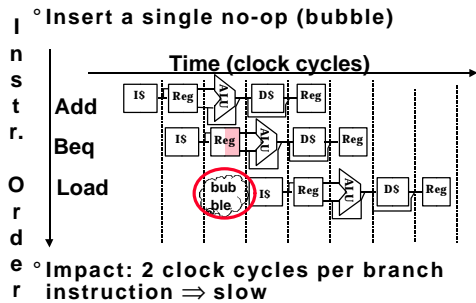
- Optimization #1:

- move comparator up to Stage 2
- as soon as instruction is decoded (Opcode identifies if 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 L24 Review Pipeline © UC Regents

18

Control Hazard: Branching (4/6)



Forwarding and Moving Branch Decision

- Forwarding/bypassing currently affects Execution stage:
 - Instead of using value from register read in Decode Stage, use value from ALU output or Memory output
- Moving branch decision from Execution Stage to Decode Stage means forwarding /bypassing must be replicated in Decode Stage for branches. I.e., Code below must still work:

```
addiu $s1, $s1, -4
beq $s1, $s2, Exit
```

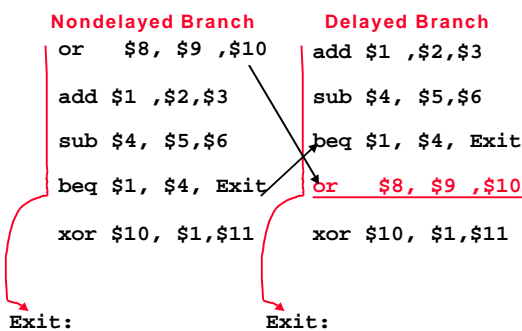
Control Hazard: Branching (5/6)

- 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**)

Control Hazard: Branching (6/6)

- 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

Example: Nondelayed vs. Delayed Branch



Try "Peer-to-Peer" Instruction

- Given question, everyone has one minute to pick an answer
- First raise hands to pick
- Then break into groups of 5, talk about the solution for a few minutes
- Then vote again (each group all votes together for the groups choice)
- discussion should lead to convergence
- Give the answer, and see if there are questions

Will try this twice today

How long to execute?

- Assume delayed branch, 5 stage pipeline, forwarding/bypassing, interlock on unresolved load hazards

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

- How many clock cycles on average to execute this code per loop iteration?
a) <= 5 b) 6 c) 7 d) 8 e) >=9
- (after 1000 iterations, so pipeline is full)

How long to execute?

- Assume delayed branch, 5 stage pipeline, forwarding/bypassing, interlock on unresolved hazards

- Look at this code:

```
Loop: 1. lw    $t0, 0($s1)
      2. addiu $t0, $t0, $s2
      3. sw    $t0, 0($s1)
      4. addiu $s1, $s1, -4
      5. bne  $s1, $zero, Loop
      6. nop
```

- How many clock cycles to execute this code per loop iteration?
a) <= 5 b) 6 c) 7 d) 8 e) >=9

Rewrite the loop to improve performance

- Rewrite this code to reduce 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 clock cycles to execute your revised code per loop iteration?
a) 4 b) 5 c) 6 d) 7

Rewrite the loop to improve performance

- Rewrite this code to reduce clock cycles per loop to as few as possible:

```
Loop: 1. lw    $t0, 0($s1)
      2. addiu $s1, $s1, -4
      3. addu  $t0, $t0, $s2
      4. bne  $s1, $zero, Loop
      5. sw    $t0, +4($s1)
```

- How many clock cycles to execute your revised code per loop iteration?
a) 4 b) 5 c) 6 d) 7

State of the Art: Pentium 4

- 1 8KB Instruction cache, 1 8 KB Data cache, 256 KB L2 cache on chip
- Clock cycle = 0.67 nanoseconds, or 1500 MHz clock rate (667 picoseconds, 1.5 GHz)
- HW translates from 80x86 to MIPS-like micro-ops
- 20 stage pipeline
- Superscalar: fetch, retire up to 3 instructions /clock cycle; Execution out-of-order
- Faster memory bus: 400 MHz

Things to Remember (1/2)

- Optimal Pipeline

- Each stage is executing part of an instruction each clock cycle.
- One instruction finishes during each clock cycle.
- On average, execute far more quickly.

- What makes this work?

- 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.

Things to Remember (2/2)

◦ **Pipelining a Big Idea: widely used concept**

◦ **What makes it less than perfect?**

- **Structural hazards: suppose we had only one cache?**

⇒ **Need more HW resources**

- **Control hazards: need to worry about branch instructions?**

⇒ **Delayed branch or branch prediction**

- **Data hazards: an instruction depends on a previous instruction?**