CS 61C: Great Ideas in Computer Architecture *Pipelining*

Instructor:
David A. Patterson

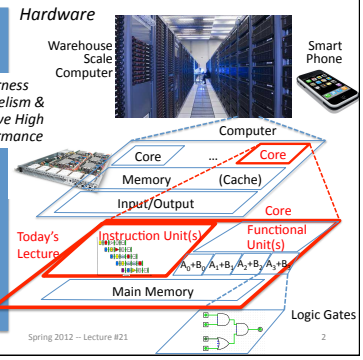
http://inst.eecs.Berkeley.edu/~cs61c/sp12

You Are Here!

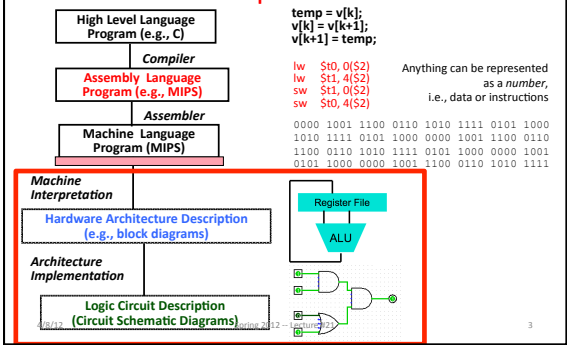
Software | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance



Levels of Representation/ Interpretation



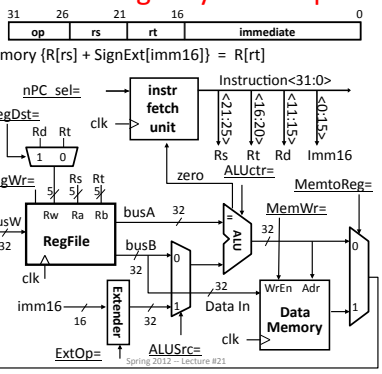
Agenda

- Pipelined Execution
- Administrivia
- Pipelined Datapath
- Pipeline Hazards
- Summary

Review: Pipelining Lessons

- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages (4 in this case)
- Time to fill pipeline and time to drain it reduces speedup: 8 hours/3.5 hours or 2.3X v. potential 4X in this example
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup

Review: Single Cycle Datapath

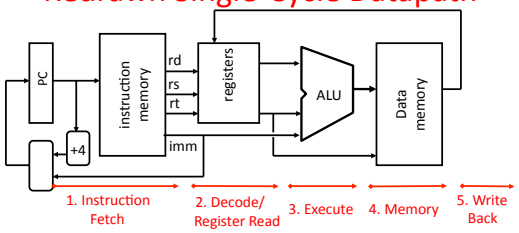


- Data Memory {R[rs] + SignExt[imm16]} = R[rt]

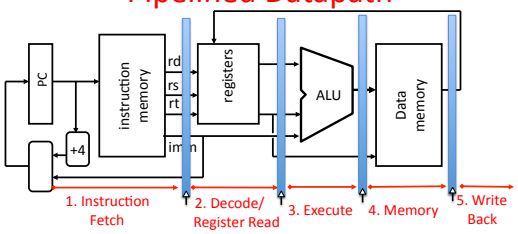
Steps in Executing MIPS

1) IF: Instruction Fetch, Increment PC
2) ID: Instruction Decode, Read Registers
3) EX: Execution
   Memory-reference: Calculate Address
   Arithmetic-logic: Perform Operation
4) Mem:
   Load: Read Data from Memory
   Store: Write Data to Memory
5) WB: Write Data Back to Register

Redrawn Single-Cycle Datapath



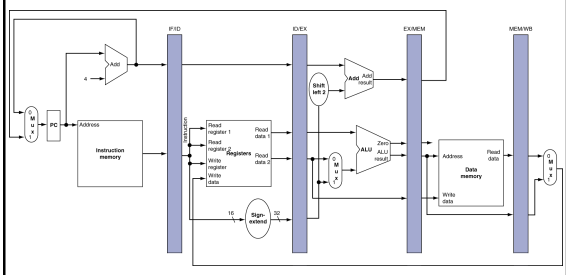
Pipelined Datapath



- Add registers between stages
  - Hold information produced in previous cycle
- 5 stage pipeline; clock rate potential 5X faster

More Detailed Pipeline

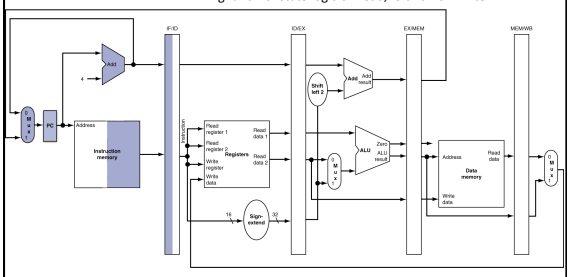
Registers named for adjacent stages, e.g., IF/ID



IF for Load, Store, …

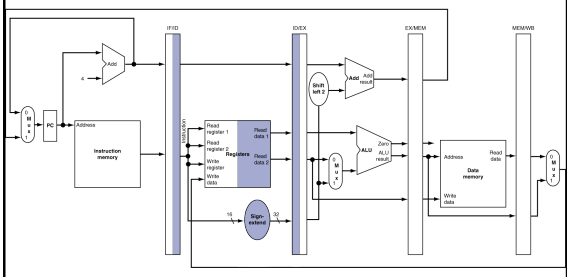
lw
Instruction fetch

Highlight combinational logic components used
+ right half of state logic on read, left half on write

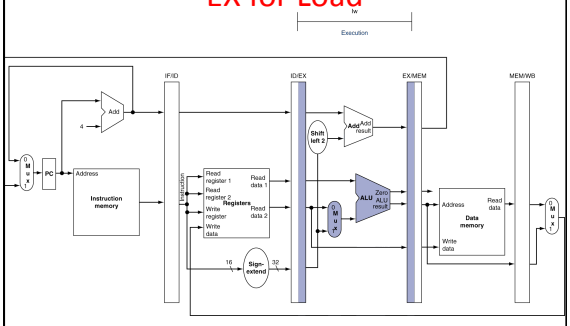


ID for Load, Store, …

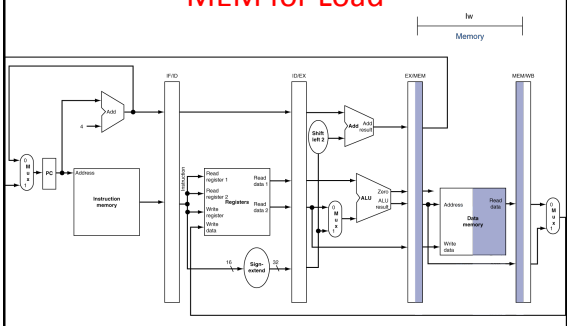
lw
Instruction decode



EX for Load

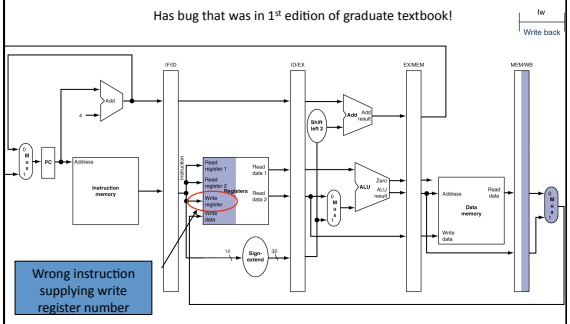


MEM for Load

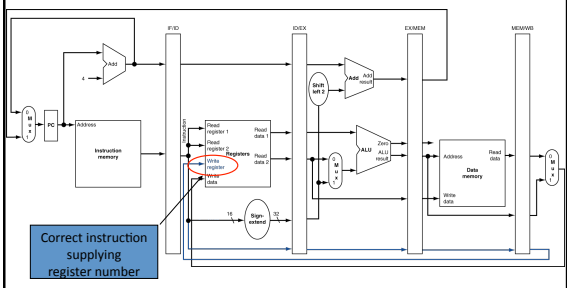


WB for Load

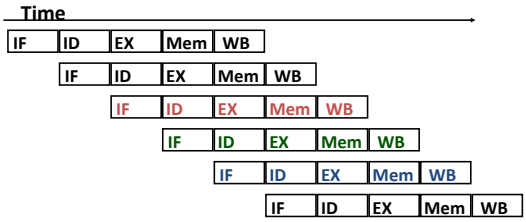
Has bug that was in 1st edition of graduate textbook!



Corrected Datapath for Load

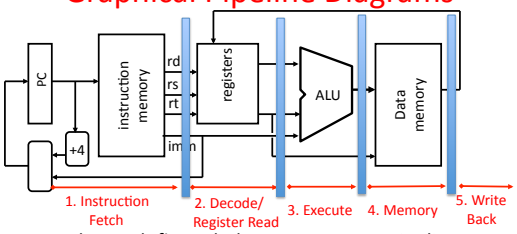


Pipelined Execution Representation

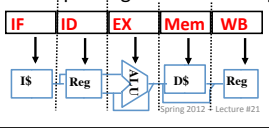


- Every instruction must take same number of steps, also called pipeline stages, so some will go idle sometimes

Graphical Pipeline Diagrams

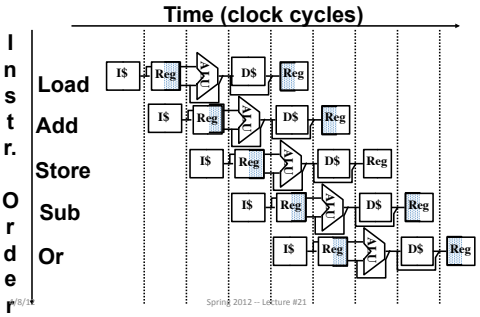


- Use datapath figure below to represent pipeline



Graphical Pipeline Representation

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



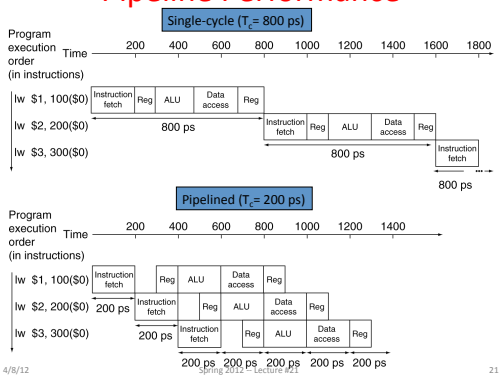
Pipeline Performance

- Assume time for stages is
  - 100 ps for register read or write
  - 200 ps for other stages
- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
sw	200 ps	100 ps	200 ps	200 ps		700 ps
R-format	200 ps	100 ps	200 ps		100 ps	600 ps
beq	200 ps	100 ps	200 ps			500 ps

Student Roulette?

Pipeline Performance



Administrivia

- Project 4: Pipelined Cycle Processor in Logicsim
  - due 4/15 (No Part 1)
- Extra Credit: Fastest Version of Project 3
  - Due 4/22 11:59 PM
- Final Review: April 29, 2PM-5PM, 2050 VLSB
- All grades finalized: 4/27
- Final: Wednesday May 9 11:30-2:30 (TBD)

Getting to Know Your Prof

- Pattersons are Scotch-Irish
  - Raised Presbyterian, which is Church of Scotland
  - Came to Pennsylvania ~1800
- We wear kilts to weddings
  - Sporran for storage
  - Sgian Dubh ("ski en doo")
- Tartan Day April 5
  - Declaration of Arbroath 1320 => Declaration of Independence
  - See movie "Braveheart" 1995

"...for, as long as but a hundred of us remain alive, never will we on any conditions be brought under English rule. It is in truth not for glory, nor riches, nor honours that we are fighting, but for freedom – for that alone, which no honest man gives up but with life itself."

Agenda

- Pipelined Execution
- Administrivia
- Pipelined Datapath
- Pipeline Hazards
- Summary

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Instruction Level Parallelism (ILP)

- Another parallelism form to go with Request Level Parallelism and Data Level Parallelism
  - RLP – e.g., Warehouse Scale Computing
  - DLP – e.g., SIMD, Map-Reduce
- *ILP – e.g., Pipelined Instruction Execution*
  - 5 stage pipeline => 5 instructions executing simultaneously, one at each pipeline stage

Hazards

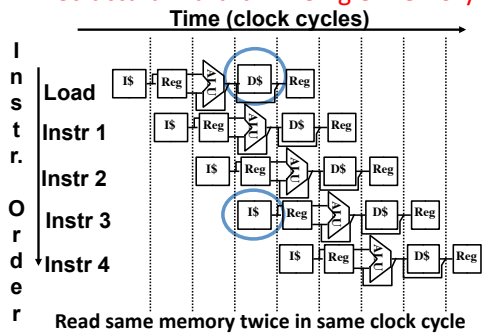
Situations that prevent starting the next logical instruction in the next clock cycle

1. Structural hazards
   - Required resource is busy (e.g., stasher is studying)
2. Data hazard
   - Need to wait for previous instruction to complete its data read/write (e.g., pair of socks in different loads)
3. Control hazard
   - Deciding on control action depends on previous instruction (e.g., how much detergent based on how clean prior load turns out)

1. Structural Hazards

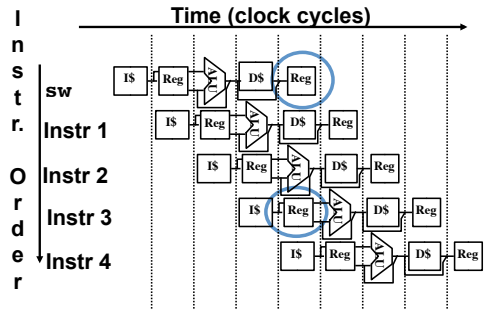
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/Store requires memory access for data
  - Instruction fetch would have to *stall* for that cycle
    - Causes a pipeline "*bubble*"
- Hence, pipelined datapaths require separate instruction/data memories
  - In reality, provide separate L1 instruction cache and L1 data cache

1. Structural Hazard #1: Single Memory



Read same memory twice in same clock cycle

1. Structural Hazard #2: Registers (1/2)



Can we read and write to registers simultaneously?

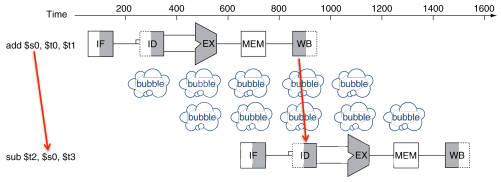
1. Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) RegFile access is *VERY* fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle
  2) Build RegFile with independent read and write ports
- Result: can perform Read and Write during same clock cycle

2. Data Hazards

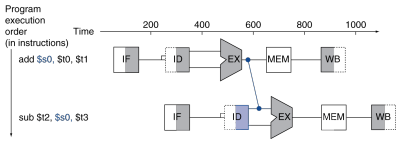
- An instruction depends on completion of data access by a previous instruction

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```

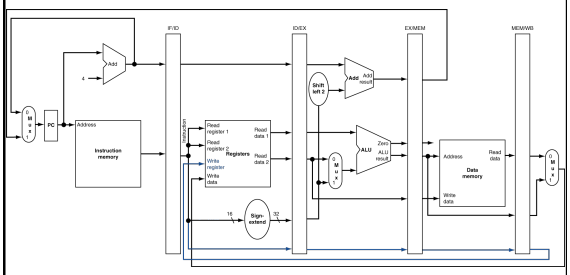


Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath

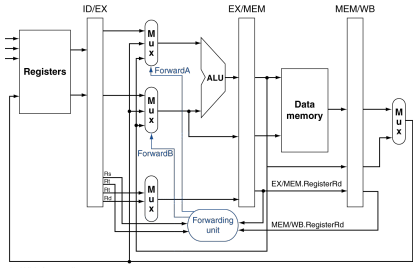


Corrected Datapath for Forwarding?



Student Roulette?

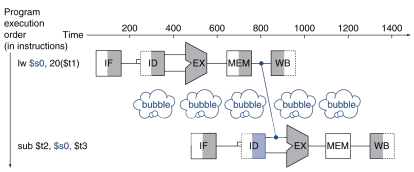
Forwarding Paths



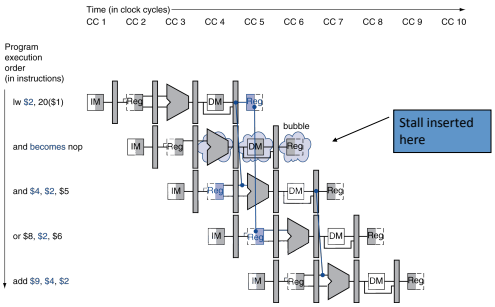
b. With forwarding

Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't bypass backward in time!



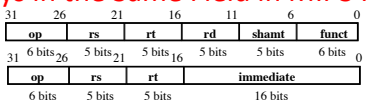
Stall/Bubble in the Pipeline



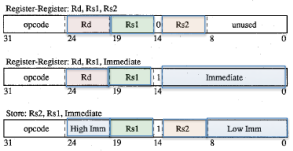
Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - x86: 1- to 17-byte instructions
  (x86 HW actually translates to internal RISC instructions!)
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step
- Memory operands only in Loads and Stores
  - Can calculate address 3rd stage, access memory 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle

Why Isn't the Destination Register Always in the Same Field in MIPS ISA?



- Need to have 2 part immediate if 2 sources and 1 destination always in same place

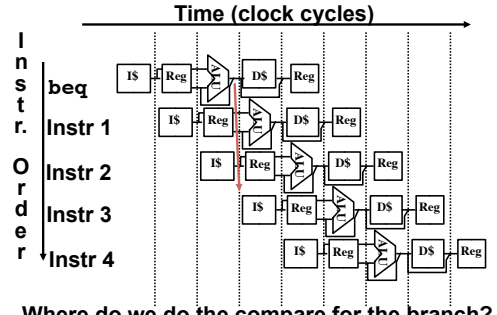


SPUR processor (A project Dave and Randy Katz worked on together)

3. Control Hazards

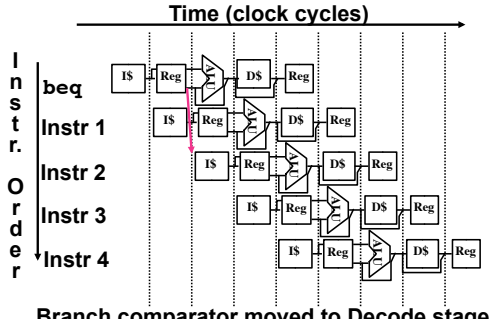
- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- BEQ, BNE in MIPS pipeline
- Simple solution Option 1: *Stall* on every branch until have new PC value
  - Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)

Stall => 2 Bubbles/Clocks



Where do we do the compare for the branch?

One Clock Cycle Stall



Branch comparator moved to Decode stage.

Code Scheduling to Avoid Stalls

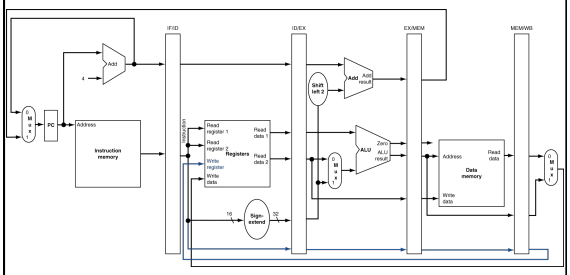
- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`

```
stall   lw  $t1, 0($t0)        lw  $t1, 0($t0)
        lw  $t2, 4($t0)        lw  $t2, 4($t0)
        add $t3, $t1, $t2      lw  $t4, 8($t0)
        sw  $t3, 12($t0)       add $t3, $t1, $t2
        lw  $t4, 8($t0)        sw  $t3, 12($t0)
stall   add $t5, $t1, $t4      add $t5, $t1, $t4
        sw  $t5, 16($t0)       sw  $t5, 16($t0)
        13 cycles              11 cycles
```


3. Control Hazard: Branching

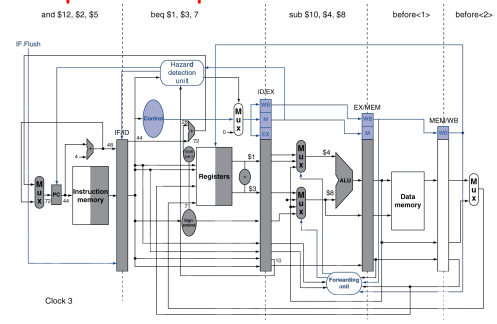
- Optimization #1:
  - Insert special branch comparator in Stage 2
  - As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only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

Corrected Datapath for BEQ/BNE?



Student Roulette?

Improved Pipeline Branch Taken



3. Control Hazards

- Option 2: *Predict* outcome of a branch, fix up if guess wrong
  - Must cancel all instructions in pipeline that depended on guess that was wrong
- Simplest hardware if we predict that all branches are NOT taken
  - Why?

Student Roulette?

3. Control Hazard: Branching

- Option #3: 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 (the *branch-delay slot*)
- *Delayed Branch* means *we always execute inst after branch*
- This optimization is used with MIPS

3. Control Hazard: Branching

- Notes on Branch-Delay Slot
  - Worst-Case Scenario: put a no-op in the branch-delay slot
  - Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn't affect the logic of program
    - Re-ordering instructions is common way to speed up programs
    - Compiler usually finds such an instruction 50% of 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:`

Delayed Branch/Jump and MIPS ISA?

- Why does JAL put PC+8 in register 31?

Student Roulette?

Delayed Branch/Jump and MIPS ISA?

- Why does JAL put PC+8 in register 31?
- *JAL executes following instruction (PC+4) so should return to PC+8*

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

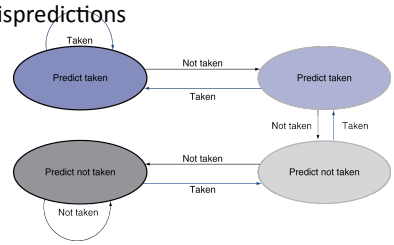
- Inner loop branches mispredicted twice!

```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions



And in Conclusion, …

The BIG Picture

- Pipelining improves performance by increasing instruction throughput: exploits ILP
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure