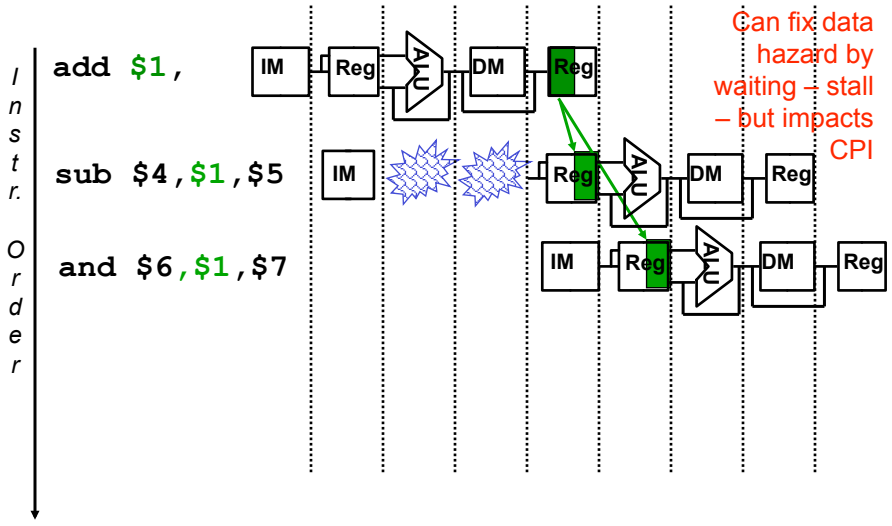
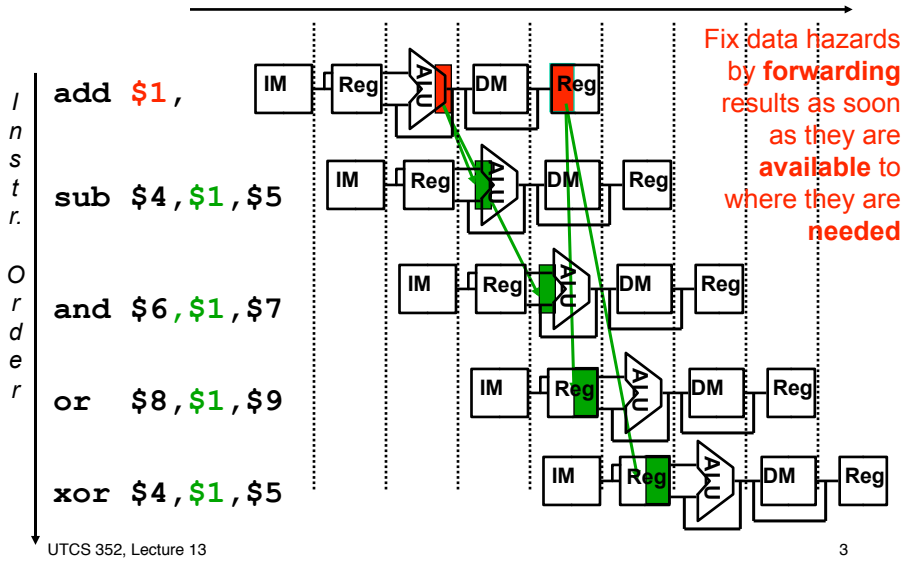
Lecture 13: Pipelined Processor

- Last time
  - Pipelining in the real world
  - Data hazards
- Today
  - Take QUIZ 9 over P&H 4.8-9, before 11:59pm today
  - Homework 4 due today
  - Homework 5 due Thursday March 11, 2010
  - Control hazards
  - Pipelining in other worlds

One Way to "Fix" a Data Hazard



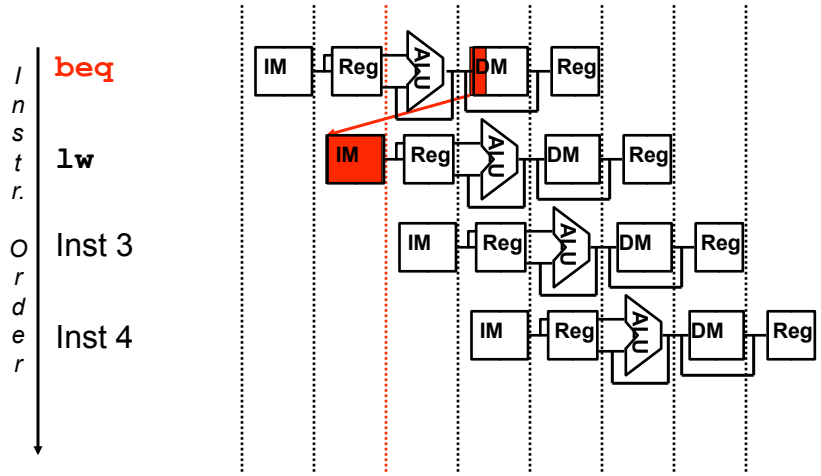
Another Way to "Fix" a Data Hazard



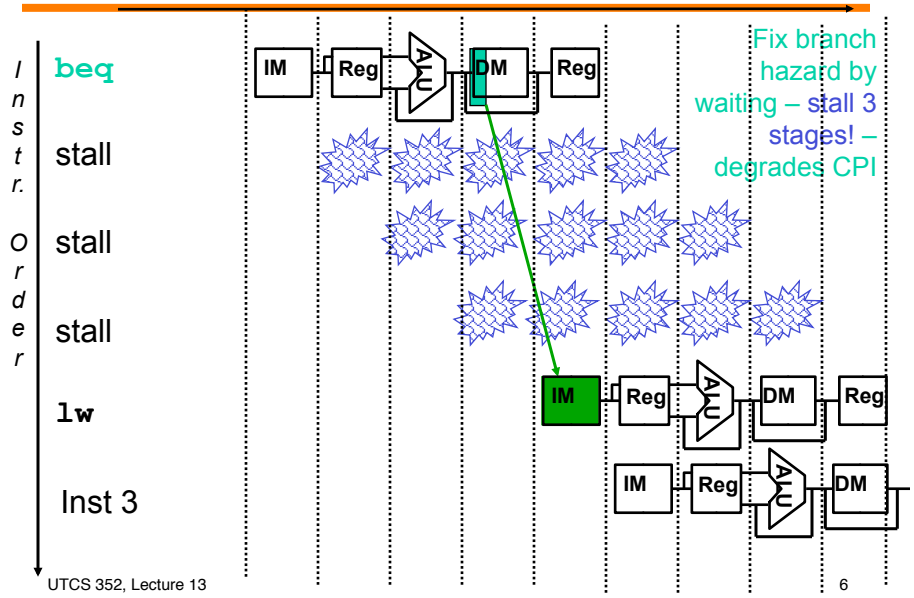
Fix data hazards by **forwarding** results as soon as they are **available** to where they are **needed**

Control Hazards

Control Hazards



One Way to "Fix" a Control Hazard



Branch Delay Slots

- Since we need to have a dead cycle anyway, let's put a useful instruction there

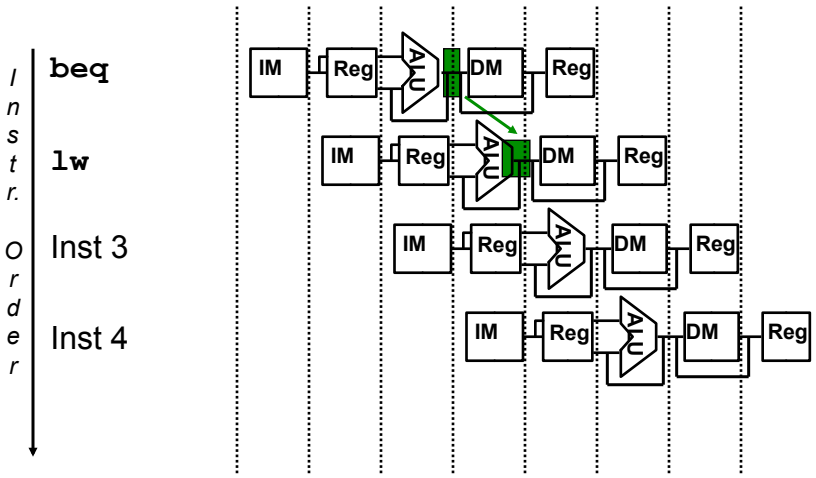
- Advantage:
  - Do more useful work
  - Potentially get rid of all stalls

- Disadvantage:
  - Exposes microarchitecture to ISA
  - Deeper pipelines require more delay slots

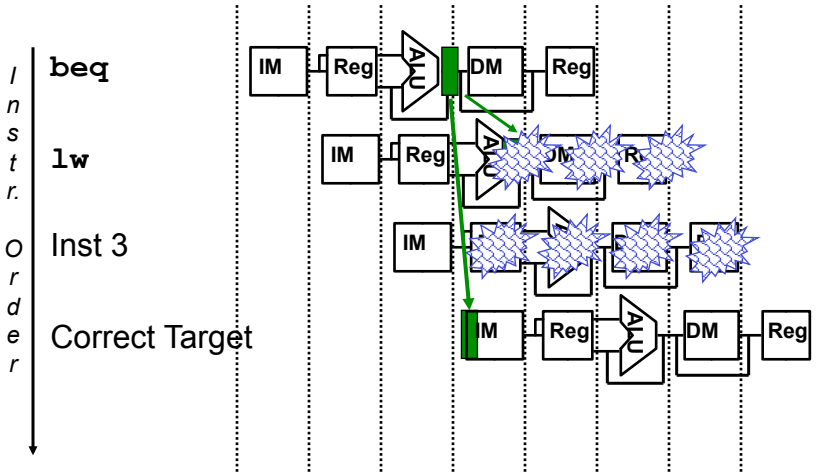
```
ADD R2,R3,R4
BNEZ R5,_loop
NOP
```

```
BNEZ R5,_loop
ADD R2,R3,R4
```

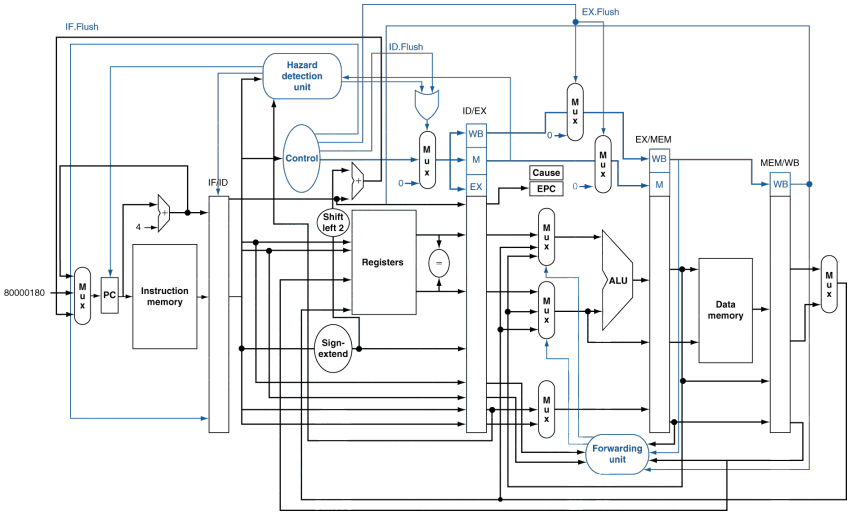

Speculate Correctly: do nothing (not quite)



Speculate Incorrectly: squash speculative instructions



Delay speculative instruction write back & flush on mispredict



Branching Structures

- Predict not taken works well for "top of the loop" branching structures
  - But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead

```
Loop: beq $1,$2,Out
      1nd loop instr
            .
            .
            .
      last loop instr
      j  Loop
Out:  fall out instr
```

- Predict not taken doesn't work well for "bottom of the loop" branching structures

```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```


Static Branch Prediction, con't

- Resolve branch hazards by assuming a given outcome and proceeding

- Predict not-taken - easiest
  - We have already fetched the fall-through instruction
  - We don't need to compute a target address
- Predict taken – predict branches will always be taken
  - Predict taken *always* incurs one stall cycle (if branch destination hardware has been moved to the ID stage)
  - Is there a way to "cache" the address of the branch target instruction ??
- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance.

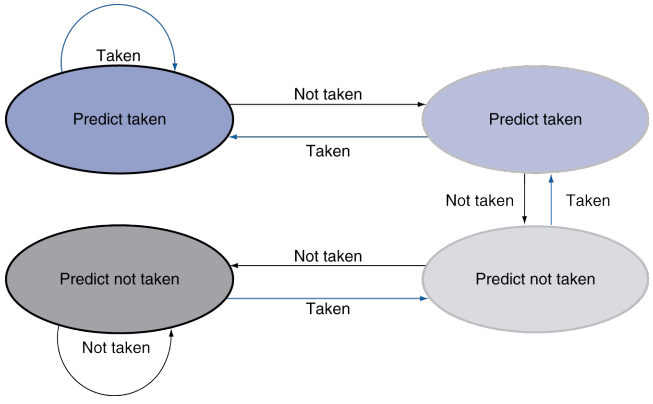
Dynamic Branch Prediction

- History: use the past branch behavior to predict the future
  - Last time
  - Last two times (why is this useful?)
- Past history of last several branches
  - Why is this useful?

```
for (i=0; i<1000; i++) {
    for (j=0; j<1000; j++) {
       if (j == 0) {
             foo;
       }
        bar;
    }
}
```

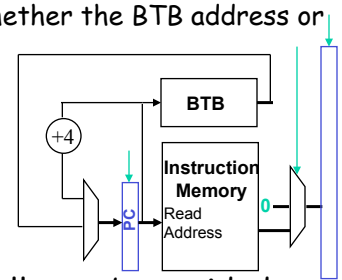

Example : 2-Bit Predictor

- Only change prediction on two successive mispredictions



Branch Target Buffer

- The predictor predicts *when* a branch is taken, but does not tell *where* its taken to!
- A branch target buffer (BTB) in the IF stage can cache the branch target address
  - The branch predictor controls whether the BTB address or PC+4 is loaded back into the PC



- If the prediction is correct, stalls can be avoided no matter which direction they go

Summary of Basic Prediction

- Conventional architectures use branch prediction to solve the "fetch" problem
  - Direction Prediction – branch taken/not taken?
  - Target Prediction – address of next instruction?
- Branch predictors use the past history of branches to predict future branches
  - Current branch's direction can depend on the past history of this branch's behavior (local)
  - Current branch's direction can depend on the direction of the last *n* branches that were encountered before this branch (global)
    - Suppose pattern is TNNTNNTNN…. how to predict this?

Branch Prediction Performance Analysis

- How important is branch prediction?
  - For IA32, about 8-16% of instructions are branches in SPECcpu2000 (C/C++), Java DaCapo, SPECjvm, but Java has more indirect branches
  - or in other words: every ~8th instruction is a branch!
- Assume balanced pipeline depth of 5, 1 cycle/stage
  - Perfect branch prediction, I & D cache, forwarding, etc.
    - CPI = 1
  - No branch prediction: stall of 3,
    - CPI = 1.375 = 11/8

      8 instructions take 11 cycles to execute on average
      - 4 cycles to fill pipeline
      - 4 cycles when 1 instruction completes per cycle
      - 3 stall cycles at branch

Branch Prediction Performance Analysis

- How important is branch prediction?
  - For IA32, about 8-16% of instructions are branches in SPECcpu2000 (C/C++), Java DaCapo, SPECjvm, but Java has more indirect branches
  - or in other words: every ~8th instruction is a branch!
- Assume balanced pipeline depth of 5, 1 cycle/stage
  - Perfect branch prediction, I & D cache, forwarding, etc.
    - CPI = 1
  - No branch prediction: stall of 3,
    - CPI =

Branch Prediction Performance Analysis

- Assume 5 stage pipeline
  - Perfect branch prediction, I & D cache, forwarding, etc.
    - CPI = 1
  - No branch prediction: stall of 3
    - CPI = 1.375 = 11/8
  - Branch prediction with 80% accuracy
    - a miss prediction every ?? instructions
    - CPI =

Branch Prediction Performance Analysis

- Assume 5 stage pipeline
  - Perfect branch prediction, I & D cache, forwarding, etc.
    - CPI = 1
  - No branch prediction: stall of 3
    - CPI = 1.375 = 11/8
  - Branch prediction with 80% accuracy
    - a miss prediction every ~42 instructions 1/(.12 * .8)
    - CPI = 1.11 = 47/42
      - 4 cycles to ramp up
      - 38 cycles executing one instruction per cycle
      - ~5 cycles to flush miss predicted instructions
        - » Notice miss prediction penalty higher than stalling

Branch Prediction Performance Analysis

- Branch Prediction crucial to modern processors!
- Flush cost is a function of
  - Pipeline depth
  - Stages between prediction & resolution
  - time to clear pipeline (number of squashed instructions)
  - time to fetch correct instruction (may miss in I-cache)
  - time to ramp up pipeline

Control Hazards Summary

- Three approaches
  - Stall until new PC is known
  - Speculate that branch goes a particular way
    - If guess is right, great!
    - If guess is wrong, kill off speculated work
  - Delay slot
- Delay slot is only approach visible to programmer!
  - Unfortunately, MIPS picked this approach!

Other Ways to Speed up the Pipeline?

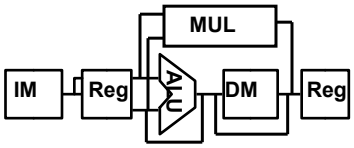
- Pipeline too long ⇒ more ALUs (exploit ILP)
- WAR/WAW hazards ⇒ register renaming

```
ADD R1,R2,R3          ADD R1,R2,R3
SUB R1,R4,R5    ⇒     SUB R1',R4,R5
```

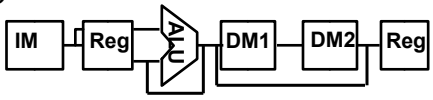
- Undetermined dependencies at compile time ⇒ dynamic scheduling
  - Object code compatibility
  - Simplify compiler
- Too many branches ⇒ better branch prediction
  - Or use predication to eliminate branches
- Unknown dependencies (control/data) ⇒ speculate
- Explicitly parallel architectures

Other Pipeline Structures Are Possible

- What about the (slow) multiply operation?
  - Make the clock twice as slow or …
  - let it take two cycles (since it doesn't use the DM stage)

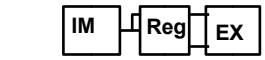


- What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)



Sample Pipeline Alternatives

- ARM7

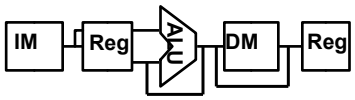


PC update
IM access

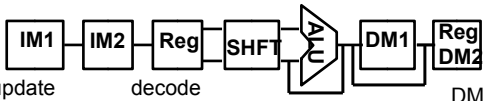
decode
reg
access

ALU op
DM access
shift/rotate
commit result
(write back)

- StrongARM-1



- XScale



PC update
BTB access
start IM access

IM access

decode
reg 1 access

shift/rotate
reg 2 access

ALU op

start DM access
exception

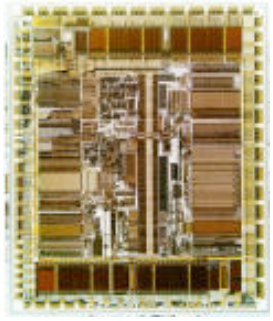
DM write
reg write

Pipelining

- All modern day processors use pipelining
- Pipelining doesn't help latency of single instruction, it helps throughput of all instructions
- Potential speedup: a CPI of 1 and fast a CC
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced pipe stages makes for inefficiencies
  - The time to "fill" pipeline and time to "drain" it can impact speedup for deep pipelines and short code runs
- Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI less than the ideal of 1)

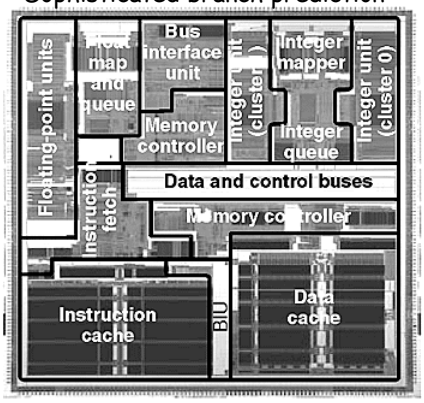
Where Are We?

Pipelined in-order processor
Simple branch prediction
Instruction/data caches (on -chip)



DEC Alpha 21064
Introduced in 1992

Out-of-order instruction execution
"Superscalar"
Sophisticated branch prediction



DEC Alpha 21264
Introduced 1998

Summary

- The real world of pipelining
  - Just stall
  - Forwarding for register and memory hazards
  - Dynamic branch prediction for control hazards
- Next Time
  - Multi-issue, Superscalar,
  - Homework 5 due Thursday March 11, 2010
- Reading: P&H 4.10-14