

EEC 485 High Performance Computer Architecture (Fall 2007)

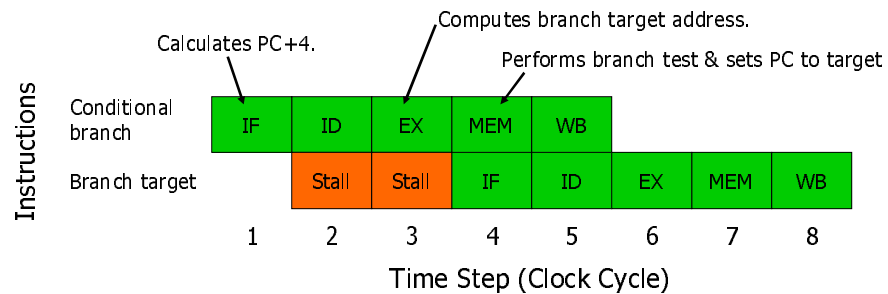
Section 6.6 Branch Hazards Section 6.8 Exceptions

Chansu Yu

Cleveland State University

Branch (Control) Hazards

While executing a previous branch, next instruction address
might not yet be known.

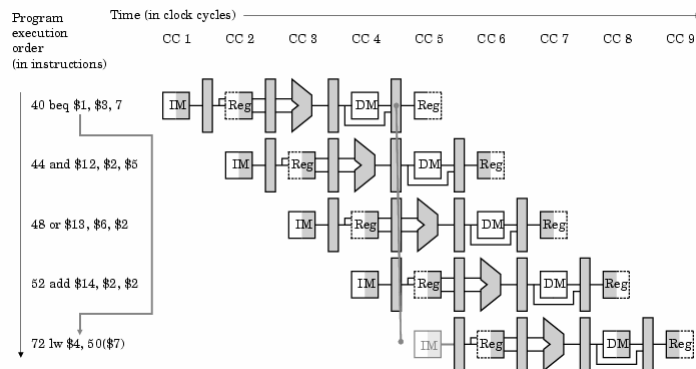


Cleveland State
University

2

c.yu91@csuohio.edu

Branch (Control) Hazards



Branch Hazards

- We can stall the pipeline for every branch instruction
 - Too slow (3 instructions)
- Or, continue execution down the sequential instruction stream assuming that the branch will not be taken (predict “branch not taken”)
 - If the condition is not met, OK ! (prediction is successful)
 - If the condition is met, (prediction is wrong)
 - Some unwanted instructions are in the pipeline!
 - Need to “flush” instructions
- How do you compare the above two ?
 - If branches are taken half the time, and if it costs little to discard the instructions, the second approach halves the cost of control hazards

Branch Hazards

- ❑ Reducing the cost of taken branch
 - Branch address procedure
 - IF: PC+4
 - EX: Branch address calculation, ZF evaluation
 - MEM: Branch target is selected
 - Selecting branch address at the ID stage to reduce the penalty to one cycle from 3 cycles
 - Branch address calculation can be done at ID stage
 - ZF evaluation: Equality can be tested at ID stage by first exclusive ORing respective bits of two read registers and then ANDing all the results
 - Control:
 - IF.Flush to flush the instruction in IF stage
 - It zeros the instruction field of the IF/ID pipeline register
 - IF.Flush = (IF/ID.Branch && ZF) ?? ← is this same as PCSrc???

IF.Flush versus Zero Control Signals

- ❑ In order to put a bubble, we nullify the control signals (for stall on a data hazard)
- ❑ Questions
 - Why can't we use the same technique for branch hazard?
 - There is no control signal at IF stage
 - Is zeroing control signals enough in case of stall?
 - As long as MemRead, MemWrite, RegWrite are not asserted, any storage value is not updated.
 - ALU will do something and MUXes will select something, but it doesn't affect any result.
 - What does it mean by flushing?
 - It zeros the instruction field of the IF/ID pipeline register, which in fact can be decoded as "sll \$0, \$0, \$0"
 - In fact, "nop" = "sll \$0, \$0, \$0"

Branch Hazards

Example

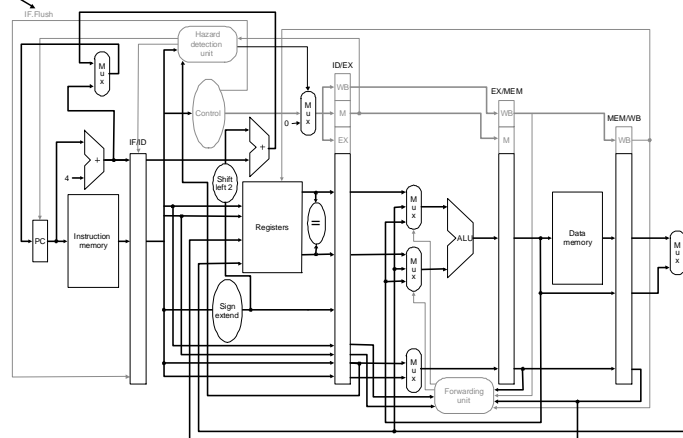
- 0040 beq \$1, \$2, 7 ; $0040+4+7*4=0072$
- 0044 and \$3, \$4, \$5
- ...
- 0072 lw \$6, 50(\$7)

branch target is calculated and ZF is checked

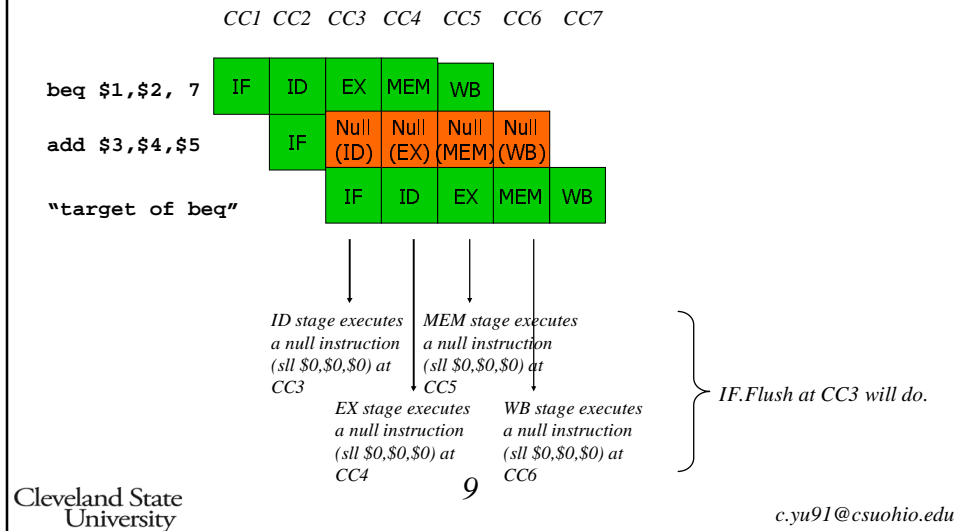
- beq IF ID EX
- and IF ---
- lw IF

Branch Hazards : Flushing

Implements flushing for branch hazards
(only one addition !) and it comes from the "Control" circuit



Stalling: What happen in the pipeline?



Branch Hazards: Improvement

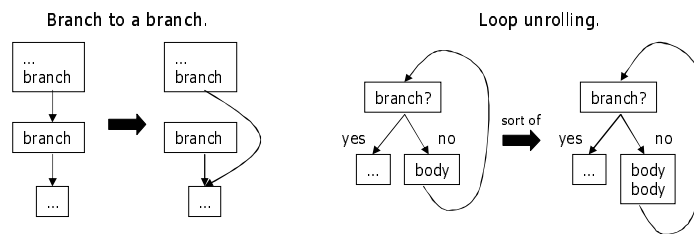
Main techniques for avoiding stalls:

- Eliminating branches
- Branch prediction
- Move comparison testing to earlier stage
- Branch delay slots

Branch Hazards: Eliminating Branches

Compiler can rewrite code to eliminate some branches.

Examples?



Branch Hazards: Earlier Branch Testing

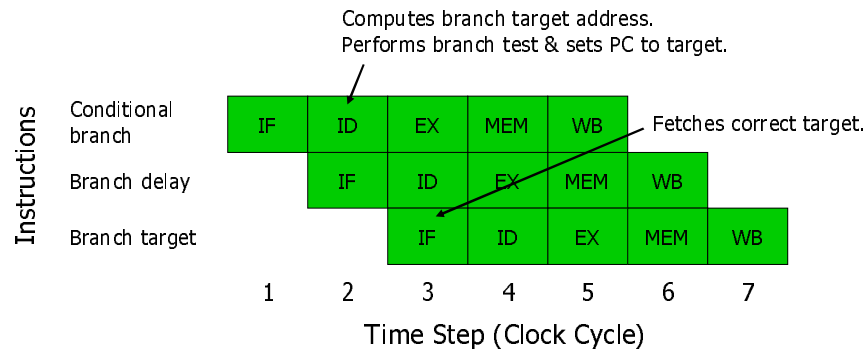
In given pipeline, tested branch conditional in EX

- Could move test to ID
 - Requires additional mini-ALU to perform tests
 - Eliminates one stall cycle
 - Could potentially increase cycle length

- Still have one cycle of stall
 - Just like unconditional branches
 - Assume this optimization

Branch Hazards: Branch Delay Slots

While determining next instruction address, go ahead and execute sequentially following instruction(s).



Branch Hazards: Branch Delay Slots

- ❑ Advantage:
 - Can avoid one stall per delay slot.
- ❑ Disadvantages:
 - Makes assembly-language programming more difficult.
 - Can be difficult to find appropriate code for slot.
 - Exposes implementation detail that could change.
 - Later implementations without a stall must still emulate slot.
- ❑ Most modern processors avoid

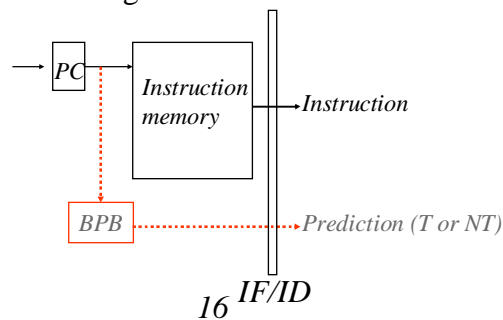
Branch Hazards: Branch Prediction

Guess which instruction is next, & start executing it.

- ❑ What if guess is wrong? : Flush the pipeline
- ❑ Simplest guesses: Always Taken or Never Taken.
- ❑ When to do prediction?
 - Static prediction: compiler
 - Dynamic prediction: processor

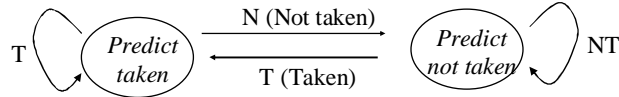
Dynamic Branch Prediction

- ❑ Branch prediction buffer (branch history table)
 - A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not.



Dynamic Branch Prediction

1-bit predictor



Prediction accuracy

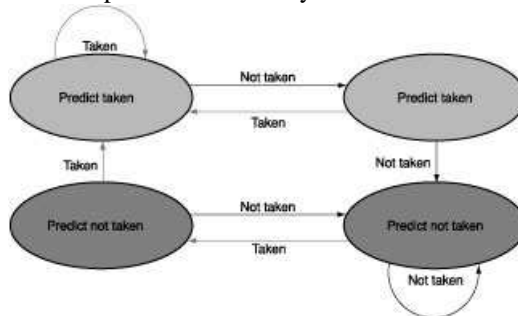
 beq) loop 10 times => 1st: ?, 2nd: correct, 3rd: correct,
 9th: correct, 10th: incorrect => 80% accuracy
 (Because the first one is incorrect in the second execution of the same code.)

17

Dynamic Branch Prediction

2-bit predictor

> What is the prediction accuracy with the same example? : 90%



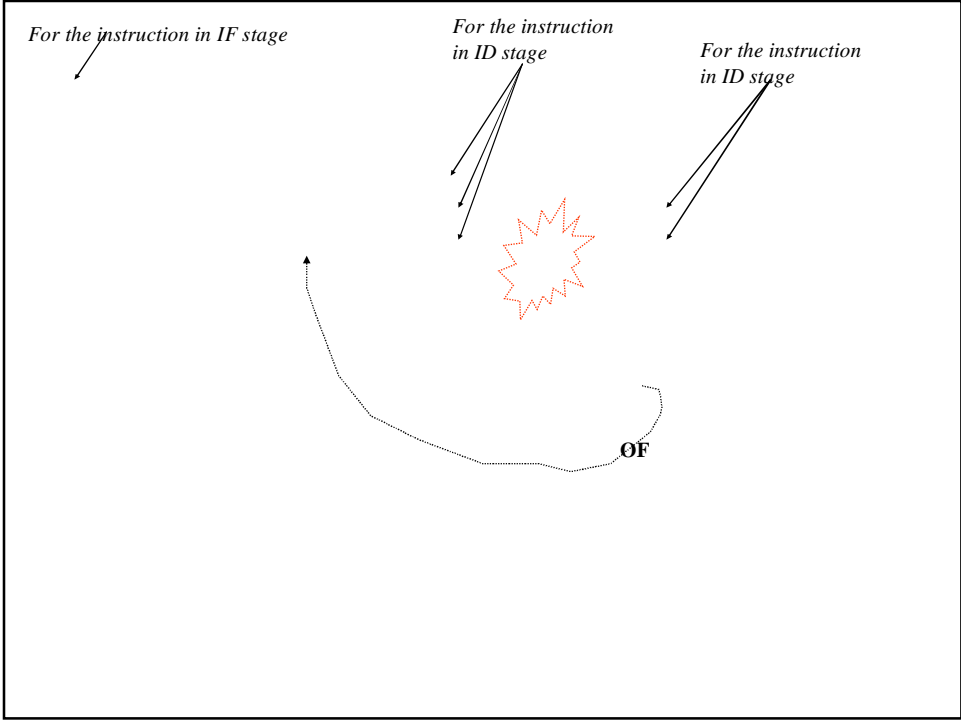
18

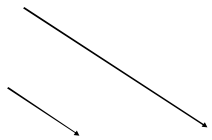
6.7 Exceptions

- ❑ Another form of control hazard involves exceptions.
- ❑ When an arithmetic overflow occurs during executing “add \$1, \$2, \$1”
 - Transfer control to the exception routine (0x4000 0040)
 - This is the same as executing a branch instruction
- ❑ For a taken-branch, we flush pipeline registers.
 - Branch is tested at the beginning of ID stage.
 - And thus flushing takes place at ID stage.
 - Since only one instruction is following after the instruction at ID, we just need to flush that instruction

Flush Control Signals

- ❑ Similar to the taken-branch, we need to flush pipeline registers. Question is which stages' pipeline register(s)?
 - Arithmetic overflow is detected at the end of EX stage.
 - And thus flushing takes place at MEM stage (at the next cycle).
 - Since three following instructions are already in the pipeline (IF, ID and EX stages), we need to flush those three instructions.
 - Otherwise, \$1 can be written back and cannot investigate the cause of the overflow.
 - Therefore, we need ID.Flush and EX.Flush in addition to IF.Flush control signal.





Challenges

- ❑ What if more than one instruction generates exceptions?
 - While “add” causes an overflow exception at CC5 in EX,
 - “lw” (with wrong opcode) causes an invalid opcode exception at CC5 at IF
- ❑ It is not OK to generate all flushing signals.
- ❑ And, how does the exception service routine correctly identify the instruction that causes the exception? => Imprecise exception

Precise and Imprecise Exceptions

- ❑ Precise exceptions
 - Hardware (CPU) correctly identifies the offending instruction.
 - And makes sure all prior instructions complete.
 - All instructions following it are not allowed to complete their execution and have not modified the process state
- ❑ Imprecise exception
 - Hardware does not guarantee it and leaves it up to the operating system to determine which instruction caused the problem.
 - Some instructions following the offending instruction are allowed to completed their execution and modified the process state.
- ❑ Most of modern CPUs support
 - Precise exceptions