
CSE 431 Computer Architecture Fall 2015

Chapter 4C: The Pipelined Processor, Part C: Dealing with Branches and Exceptions

Mary Jane Irwin (www.cse.psu.edu/~mji)

[Adapted from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, Morgan Kaufmann]

Reminders

□ This week

- Reducing branch hazard costs – PH 4.8
- Branch prediction; dealing with exceptions – PH 4.9

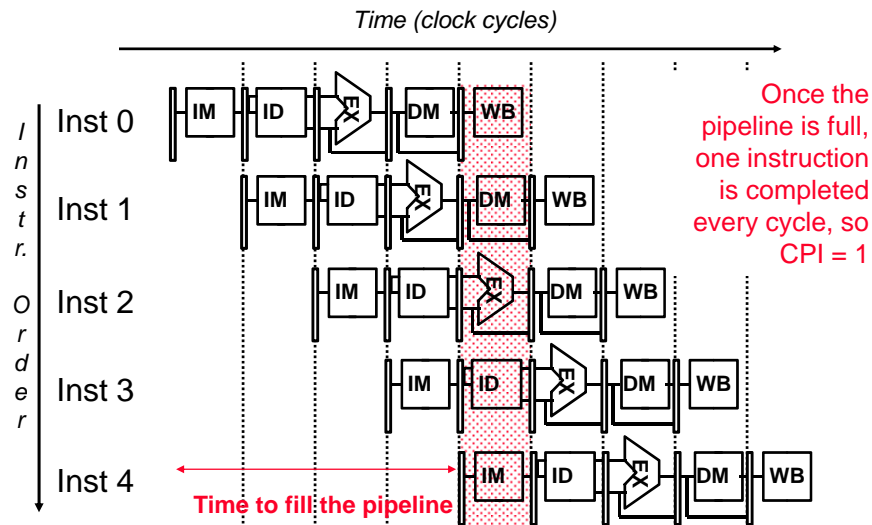
□ Next week

- Memory hierarchies, cache basics review – PH 5.1-5.2
- Improving cache performance – PH 5.3, 5.9

□ Reminders

- HW2 dropbox closes midnight Sep 17th
- HW3 will come out Sep 18th (and will be due on Sep Oct 1st)
- Quiz 2 dropbox closes midnight Sept 22nd
- Quiz 3 will open Sept 23rd (and will close midnight Oct 5th)
- First evening midterm exam scheduled
 - Tuesday, **October 6th**, 20:15 to 22:15, Location 22 Deike

Review: Why Pipeline? For Performance!



CSE431 Chapter 4B.3

Irwin, PSU, 2015

Review: Pipelining - What Makes it Hard ?

□ Pipeline Hazards

- **structural hazards**: attempt to use the same resource by two different instructions at the same time
- **data hazards**: attempt to use data before it is ready
 - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
- **control hazards**: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
 - branch and jump instructions, exceptions

- Pipeline hardware control must **detect** the hazard and then take action to **resolve** hazard

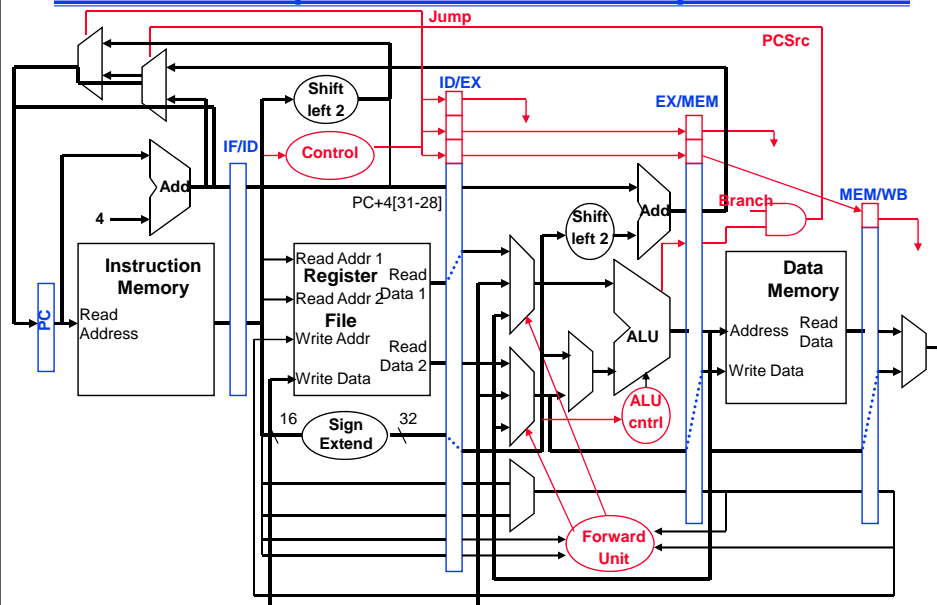
CSE431 Chapter 4B.4

Irwin, PSU, 2015

Control Hazards

- ❑ When the flow of instruction addresses is not sequential (i.e., $PC = PC + 4$); incurred by change of flow instructions
 - Unconditional branches (*j*, *jal*, *jr*)
 - Conditional branches (*beq*, *bne*)
 - Exceptions
- ❑ Possible approaches
 - Stall (impacts CPI)
 - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
 - Delay decision (requires compiler support)
 - Predict and hope for the best !
- ❑ Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards

Review: Datapath Branch and Jump Hardware



Control (Branch, Jump) Hazards

□ What do we need to know?

- Next instruction target address, maybe sequential (PC+4) or
 - `beq`, PC+4 + branch instruction's sign-extended offset which is computed during EX by the Shift Left 2 – Add logic
 - `j`, `jal`, constant address field read from IM during IF (26 bits)
 - `jr`, `jalr`, read from RF during ID (32 bits)
 - `trap` instruction or exception, obtained from table lookup in the OS (32 bits)
- Branch decision outcome (ALU zero flag)
 - continue sequentially? or jump to the branch target address?

□ When do we need to know it?

- As early as possible in the pipeline
 - if in the ID stage, at worst one stall cycle for the MIPS pipeline due to branches

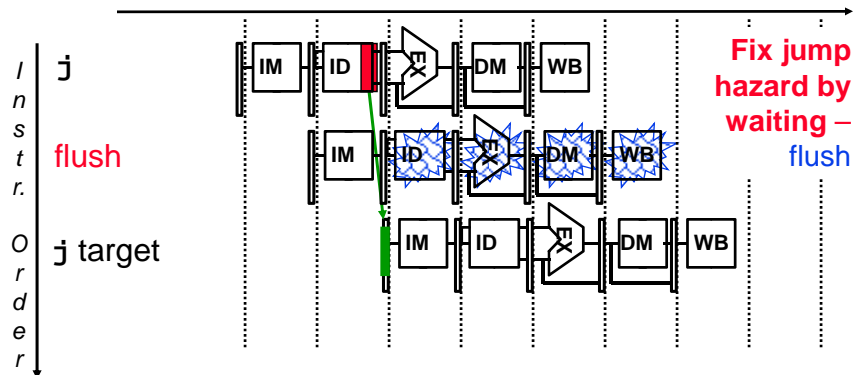
Control (Branch, Jump) Hazards, con't

□ When do we act on the decision?

- As soon as possible
 - we decided too late? We already fetched another instruction, and may need to discard it (flush it)
 - Maybe will have to flush more than one instruction?
 - we guessed, and were right – this is good
 - we guessed, and were wrong – now need to fix things
- Guesses require an evaluation of the success rate, by simulation (before the fact) or by measurement (after the fact)
 - can measurement improve the quality of the guesses?
 - recent history is often a reasonable predictor of the near future

Jumps Incur One Stall

- ❑ Jumps not decoded until ID, so one **flush** is needed
 - To flush, set `IF/ID.Flush` to zero the bits in the IF/ID pipeline register (turning it into a `noop`)

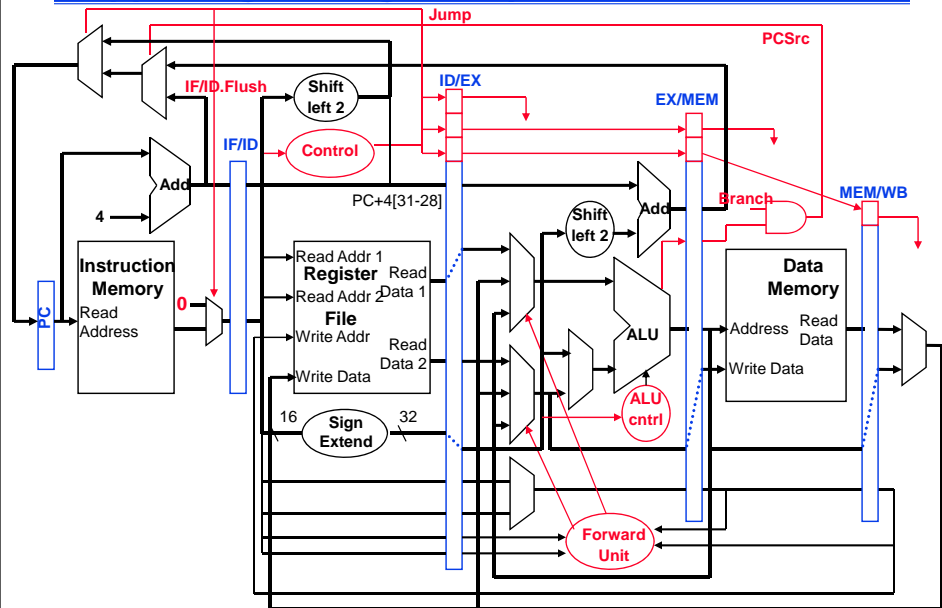


- ❑ Fortunately, jumps are very infrequent – only ~3% of the SPECint instruction mix

Two “Types” of Stalls

- ❑ **Stall** (or bubble) where a `noop` instruction is **inserted** **between** two instructions in the pipeline (as done for load-use)
 - Keep the instructions *earlier* in the pipeline (later in the code) from progressing down the pipeline for a cycle (“bounce” them in place with write control signals to the pipeline registers)
 - Insert `noop` by zeroing the control bits in the pipeline register at the appropriate stage (with `IF/ID.Bubble`)
 - Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline
- ❑ **Flush** (or instruction squashing) where an instruction in the pipeline is **replaced** with a `noop` instruction (as done for instructions located sequentially after `j` instructions with `IF/ID.Flush`)
 - Zero the control bits in the IF/ID pipeline register of the instruction to be flushed (the one just after the `j` instruction)

Supporting ID Stage Jumps

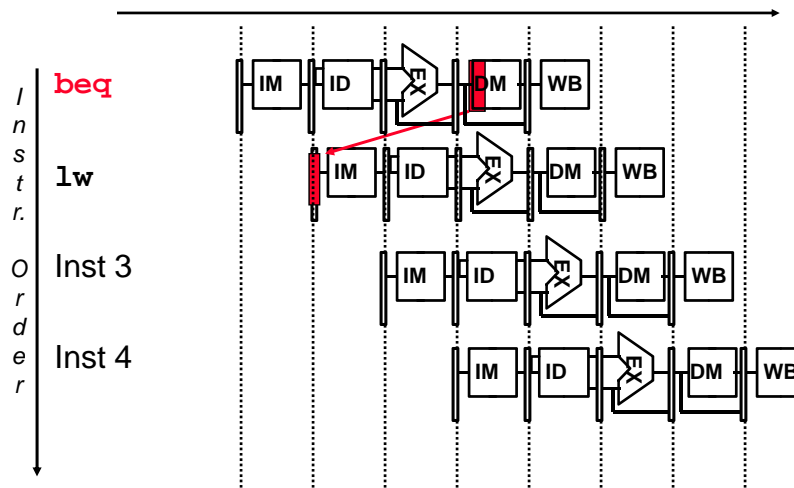


CSE431 Chapter 4B.11

Irwin, PSU, 2015

Branch Instructions Cause Control Hazards

- Dependencies backward in time cause **hazards**

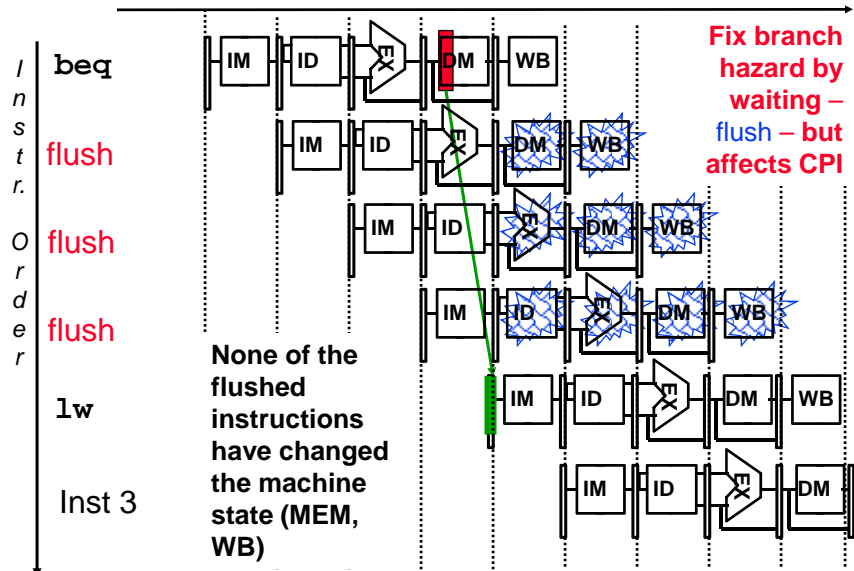


- Which instruction is executed after the branch ?

CSE431 Chapter 4B.12

Irwin, PSU, 2015

One Way to “Fix” a Branch Control Hazard

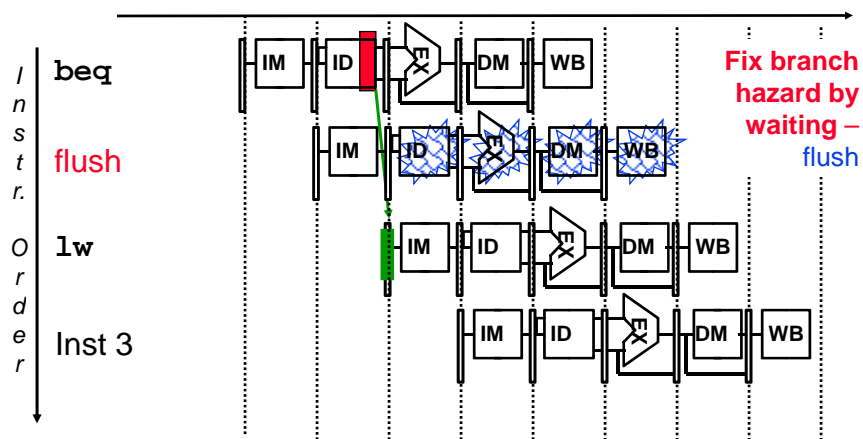


CSE431 Chapter 4B.13

Irwin, PSU, 2015

Another Way to “Fix” a Branch Control Hazard

- Move branch decision hardware back to as **early** in the pipeline as possible – i.e., during the decode cycle



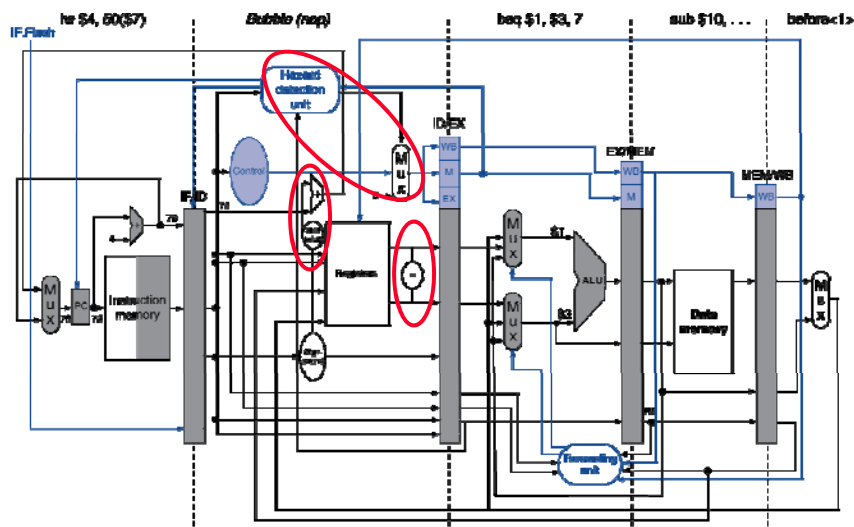
CSE431 Chapter 4B.14

Irwin, PSU, 2015

Reducing the Delay of Branches

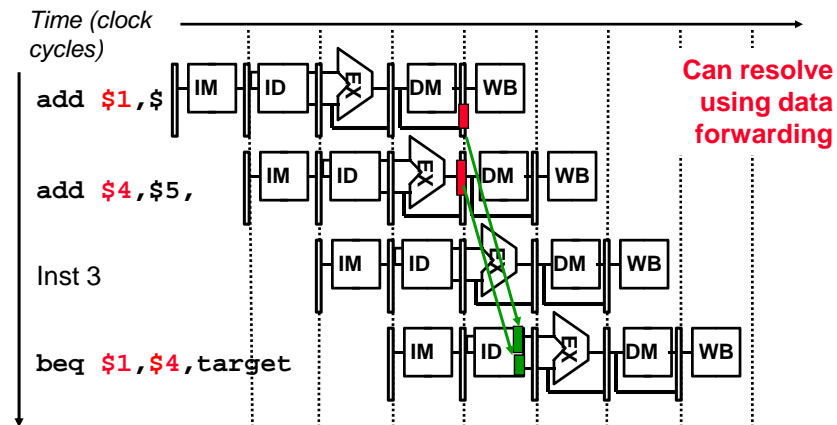
- ❑ Move the branch decision hardware back to the EX stage
 - Reduces the number of stall (flush) cycles to two
 - Adds an `and` gate and a `2x1 mux` to the EX timing path
- ❑ Move the branch decision hardware back to the ID stage
 - Reduces the number of stall (flush) cycles to one (like with jumps)
 - But now need to add **forwarding hardware** in ID stage
 - Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)
 - Comparing the registers can't be done until after RegFile read, so adds a comparator to the ID timing path
- ❑ For deeper pipelines, branch decision points can be even *later* in the pipeline, incurring more stalls

Branch Taken Example



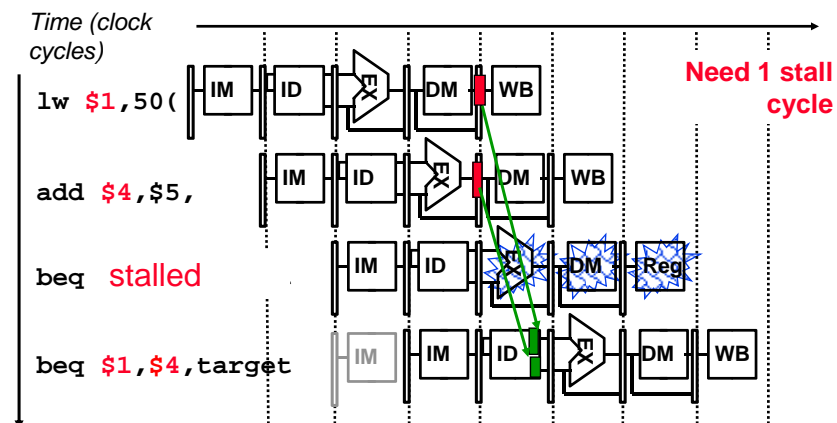
Yet Another Complication

- What if a comparison register is a destination of the 2nd or 3rd preceding ALU instruction?



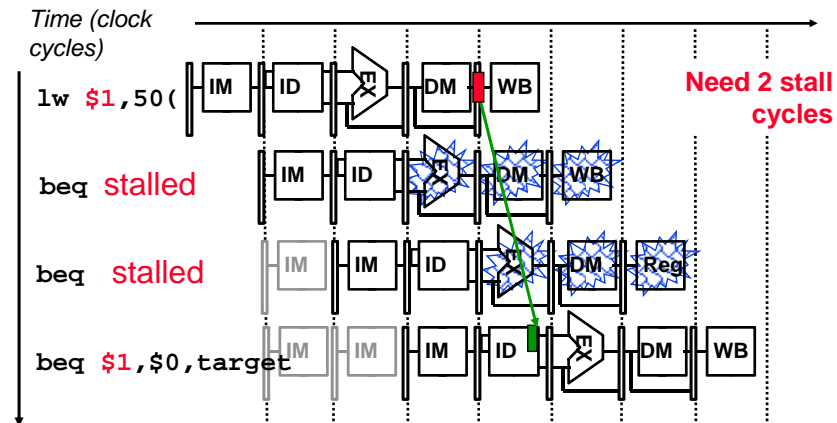
And Another Branch Data Hazard

- What if a comparison register is a destination of the preceding ALU instruction or the 2nd preceding lw instr?



And Yet Another Branch Data Hazard

- What if a comparison register is a destination of the immediately preceding preceding `lw` instr?



Static Branch Prediction

- Resolve branch hazards by **assuming** a given outcome and proceeding without waiting to see the actual branch outcome
- 1. **Predict not taken** – *always* predict branches will **not** be taken, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall
 - If taken, **flush** instructions **after** the branch (earlier in the pipeline, later in the code)
 - in IF, ID, and EX stages if branch logic in MEM – **three** stalls
 - In IF and ID stages if branch logic in EX – **two** stalls
 - in IF stage if branch logic in ID – one stall**
 - ensure that those flushed instructions haven't changed the machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))
 - restart the pipeline at the branch destination

Static Branching Structures

- Always predict NT 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 every time through the loop

```

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

- Always predict NT doesn't work well for “bottom of the loop” branching structures

- Guess wrong every time through the loop except the last time (when we fall out of the loop)

```

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

0-Bit Predictors, Loop for 10 Iterations



iteration	predict	actual	predict	actual
1	NT	T	T	T
2	NT	T	T	T
3	NT	T	T	T
4	NT	T	T	T
5	NT	T	T	T
6	NT	T	T	T
7	NT	T	T	T
8	NT	T	T	T
9	NT	T	T	T
10	NT	NT	T	NT
	10% accuracy		90% accuracy	

```

pre-loop instr
Loop:
  1st loop instr
  2nd loop instr
  .
  .
  .
  last loop instr
  bne $1,$2,Loop
  post-loop instr
  
```

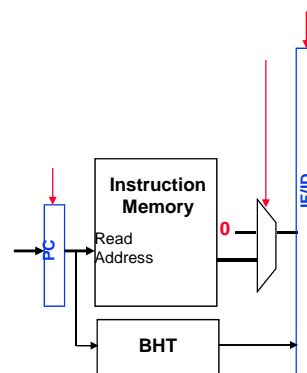
So use
predict
Taken
(T)?

Other Branch Prediction Possibilities

2. **Predict taken (T)** – predict branches will always be taken
 - BUT 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, or *better yet* the **actual** branch target **instruction** itself ?? Yes ... stay tuned !
- ❑ As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior **dynamically** during program execution
3. **Dynamic branch prediction** – predict branches at run-time using *run-time* information

Dynamic Branch Prediction Buffer

- ❑ A **branch prediction buffer** (aka branch history table (**BHT**)) in the IF stage addressed by the low order bits of the PC, contains bit(s) (passed to the ID stage through the IF/ID pipeline register) that tell whether the branch was taken or not the last time it was execute



Branch History Table

- ❑ The BHTs prediction bits may predict incorrectly (may be a wrong prediction for this branch this iteration or may be from a different branch with the same low order PC bits) but the doesn't affect **correctness**, just **performance**
 - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit(s)
- ❑ If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit(s)
- ❑ Default BHT size in SimpleScalar is 2048 entries (11 low order bits of the PC)
 - A 4096-bit BHT using 2-level adaptive prediction varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott) misprediction rate

1-bit NT Dynamic Branch Prediction Accuracy

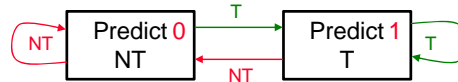
- ❑ A 1-bit NT dynamic predictor will be incorrect twice
 - Assume predict_bit = 0 (NT) to start (indicating branch not taken) and loop control is at the bottom of the loop code
 - 1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1 (T))
 - 2. As long as branch is taken (looping), prediction is correct
 - 3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0 (NT))
- ```

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

```

- ❑ For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

## 1-Bit Dynamic Predictor, Loop for 10 Iterations



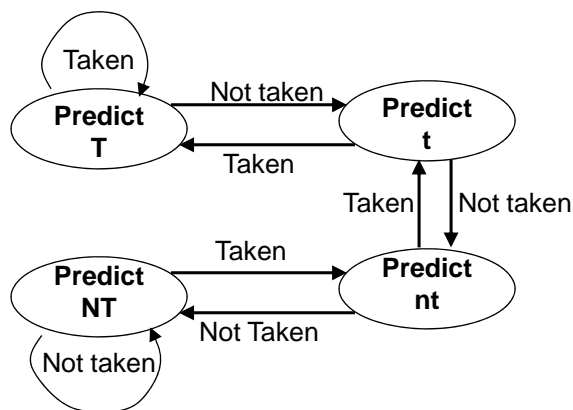
|           | Initial State = 0 |              |        | Initial State = 1 |              |        |
|-----------|-------------------|--------------|--------|-------------------|--------------|--------|
| iteration | state             | predict      | actual | state             | predict      | actual |
| 1         | 0                 | NT           | T      | 1                 | T            | T      |
| 2         | 1                 | T            | T      | 1                 | T            | T      |
| 3         | 1                 | T            | T      | 1                 | T            | T      |
| 4         | 1                 | T            | T      | 1                 | T            | T      |
| 5         | 1                 | T            | T      | 1                 | T            | T      |
| 6         | 1                 | T            | T      | 1                 | T            | T      |
| 7         | 1                 | T            | T      | 1                 | T            | T      |
| 8         | 1                 | T            | T      | 1                 | T            | T      |
| 9         | 1                 | T            | T      | 1                 | T            | T      |
| 10        | 1                 | T            | NT     | 1                 | T            | NT     |
|           | 0                 | 80% accuracy |        | 0                 | 90% accuracy |        |

CSE431 Chapter 4B.30

Irwin, PSU, 2015

## 2-bit Dynamic Branch Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed



```

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

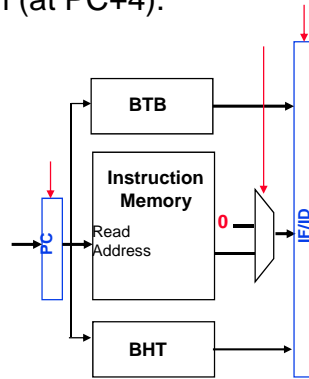
```

CSE431 Chapter 4B.31

Irwin, PSU, 2015

## But Also Must Know the Branch Target

- ❑ A **branch target buffer (BTB)** in the IF stage can cache the branch target **address**, but remember we also need to fetch the next sequential instruction (at PC+4).
  - Would need a two read port IM
- ❑ Or the BTB can cache the actual branch target **instruction** while the Instruction Memory is fetching the next sequential instruction
  - Would need a **two** read port IM
  - ID stage can then select between PC+4 and branch target instruction
- ❑ If we predict correctly, stalls can be avoided no matter which direction the branch goes



## Summary of 2-bit Dynamic Branch Predictors

- ❑ A 2-bit dynamic branch prediction scheme can give 90% accuracy since a prediction must be wrong twice before the prediction is changed
  - In a counter implementation, the counters are incremented when a branch is taken and decremented when not taken (and **saturate** at 00 or 11).
- ❑ BHT stores the initial state of the predictor's Finite State Machine (usually the last state last time through the loop)
- ❑ BTB stores the branch target instruction which is "fetched" along with the sequential instruction in the Fetch stage (when the branch is in the Decode stage)
- ❑ Since we read the prediction bits on every cycle, a 2-bit predictor will need both a read and a write access port for updating the prediction bits.

## SimpleScalar Branch Prediction

| -bpred<type> |                                                                                                        |
|--------------|--------------------------------------------------------------------------------------------------------|
| not taken    | Always predict not taken                                                                               |
| taken        | Always predict taken                                                                                   |
| perfect      | Perfect prediction (but can't build it)                                                                |
| bimod        | Bimodal predictor using a 2048 entry BHT with 2-bit counters and a BTB (512 sets, 4-way associativity) |
| 2lev         | 2-level adaptive predictor                                                                             |
| comb         | Combined predictor (bimodal and 2-level adaptive)                                                      |

- ❑ Many more possibilities, see [http://en.wikipedia.org/wiki/Branch\\_predictor](http://en.wikipedia.org/wiki/Branch_predictor)
- ❑ Prediction accuracy improves as the branch predictor grows in complexity

## Dealing with Exceptions

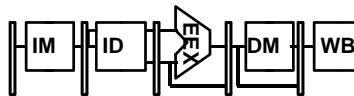
- ❑ Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  - R-type arithmetic overflow
  - Trying to execute an undefined instruction
  - An I/O device request
  - An OS service request (e.g., a page fault, TLB exception)
  - A hardware malfunction
- ❑ The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code)
- ❑ The software (OS) looks at the cause of the exception and “deals” with it



## Two Types of Exceptions

- ❑ Interrupts – asynchronous to program execution
  - caused by **external events**
  - may be handled **between** instructions, so can let the instructions currently active in the pipeline *complete* before passing control to the OS interrupt handler
  - simply suspend and resume user program
- ❑ Traps (Exception) – synchronous to program execution
  - caused by **internal events**
  - condition must be remedied by the trap handler for **that** instruction, so much stop the offending instruction *midstream* in the pipeline and pass control to the OS trap handler
  - the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted

## Where in the Pipeline Exceptions Occur



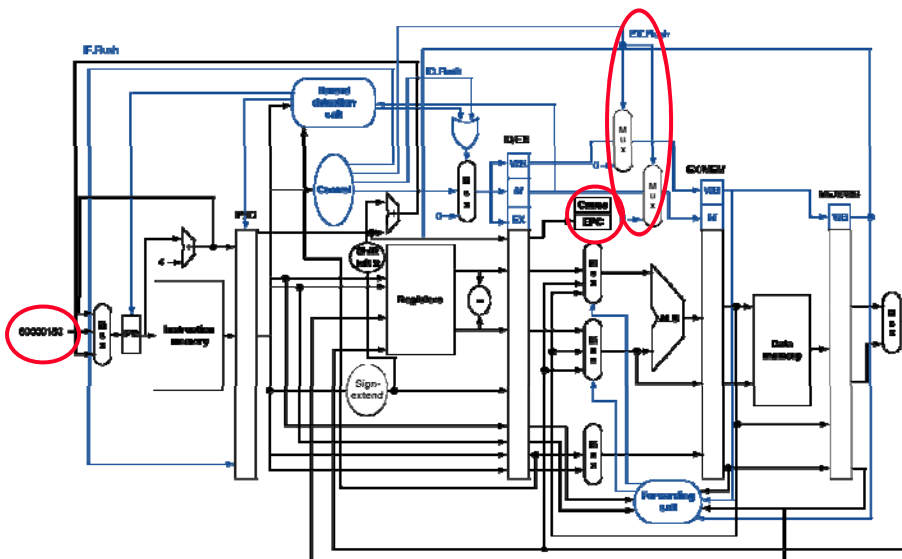
Stage(s)?      Synchronous?

- ❑ Arithmetic overflow
- ❑ Undefined instruction
- ❑ TLB or page fault
- ❑ I/O service request
- ❑ Hardware malfunction

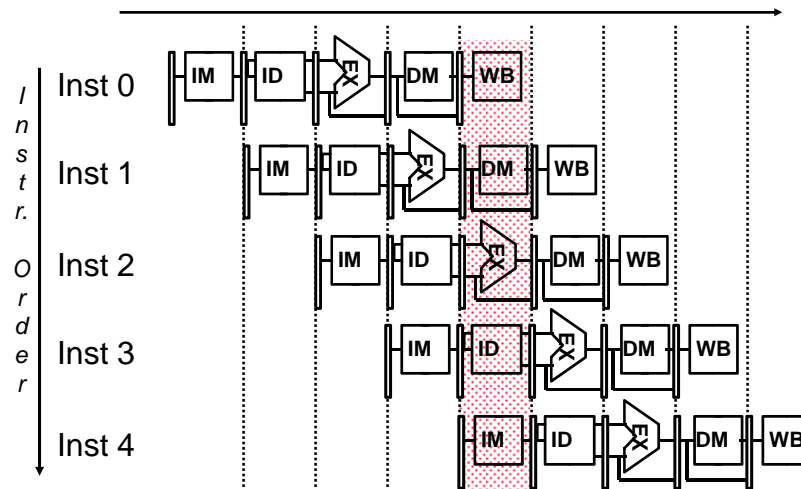
## Additions to MIPS to Handle Exceptions

- ❑ Cause register (records exceptions) – hardware to record in Cause the exceptions and a signal to control writes to it (CauseWrite)
- ❑ EPC register (records the addresses of the offending instructions) – hardware to record in EPC the address of the offending instruction and a signal to control writes to it (EPCWrite)
  - Exception software must match exception to instruction
- ❑ A way to load the PC with the address of the exception handler
  - Expand the PC input mux where the new input is hardwired to the exception handler address - (e.g., 8000 0180<sub>hex</sub> for arithmetic overflow, 8000 0000<sub>hex</sub> for undefined instruction)
- ❑ A way to flush offending instruction and the ones that follow it

## Pipeline with Exception Extensions



## Multiple Simultaneous Exceptions



- ❑ Hardware sorts the exceptions so that the earliest instruction is the one interrupted first

## Summary

- ❑ All modern day processors use pipelining for performance (a CPI of 1 and fast a CC)
- ❑ Pipeline clock rate limited by **slowest** pipeline stage – so designing a balanced pipeline is important
- ❑ Must detect and resolve hazards
  - Structural hazards – resolved by designing the pipeline correctly
  - Data hazards
    - Stall (impacts CPI)
    - Forward (requires hardware support)
  - Control hazards – put the branch decision hardware in as early a stage in the pipeline as possible
    - Stall (impacts CPI)
    - Delay decision (requires compiler support)
    - Static and **dynamic prediction** (requires hardware support)
- ❑ Pipelining complicates exception handling

## Reminders

---

### □ Next week

- Memory hierarchies, cache basics review – PH 5.1-5.2
- Improving cache performance – PH 5.3, 5.7

### □ Reminders

- HW2 dropbox closes midnight Sep 17<sup>th</sup>
- HW3 will come out Sep 18<sup>th</sup> (and will be due on Sep Oct 1<sup>st</sup>)
- Quiz 2 dropbox closes midnight Sept 22<sup>nd</sup>
- Quiz 3 will open Sept 23<sup>rd</sup> (and will close midnight Oct 5<sup>th</sup>)
- First evening midterm exam scheduled
  - Tuesday, **October 6<sup>th</sup>**, 20:15 to 22:15, Location 22 Deike