These slides are mostly taken verbatim, or with minor changes, from those prepared by Mary Jane Irwin (www.cse.psu.edu/~mji) of The Pennsylvania State University. [Adapted from Computer Organization and Design, 4th Edition, Patterson & Hennessy, © 2008, MK]
Key to the Slides

- The source of each slide is coded in the footer on the right side:
  - **Irwin CSE331** = slide by Mary Jane Irwin from the course CSE331 (Computer Organization and Design) at Pennsylvania State University.
  - **Irwin CSE431** = slide by Mary Jane Irwin from the course CSE431 (Computer Architecture) at Pennsylvania State University.
  - **Hegner UU** = slide by Stephen J. Hegner at Umeå University.
Once the pipeline is full, one instruction is completed every cycle, so CPI = 1.
Review: MIPS Pipeline Data and Control Paths

- Instruction Memory
  - Read Address
  - Write Data
- Register File
  - Read Addr 1
  - Read Addr 2
  - Write Addr
  - Read Data 1
  - Read Data 2
- Control
- Add
  - Shift left 2
  - Add
  - Branch
  - MemtoReg
  - MEM/WB
- Data Memory
  - Read Data
  - Write Data
- ALU
  - ALUSrc
  - ALU cntrl
  - ALUOp
  - RegWrite
  - MemWrite
  - MemRead
  - RegDst
  - PCSrc
  - IF/ID
  - ID/EX
  - EX/MEM
  - PCSrc
  - PC
Review: Can Pipelining Get Us Into Trouble?

- Yes: 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 control must **detect** the hazard and then take action to **resolve** hazards
Review: Register Usage Can Cause Data Hazards

- Read after write (RAW) data hazard

<table>
<thead>
<tr>
<th>Value of $1</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10/-20</th>
<th>-20</th>
<th>-20</th>
<th>-20</th>
<th>-20</th>
</tr>
</thead>
</table>

- **add** $1,$

- **sub** $4,$1,$5

- **and** $6,$1,$7

- **or** $8,$1,$9

- **xor** $4,$1,$5
One Way to “Fix” a Data Hazard

Can fix data hazard by waiting – stall – but impacts CPI

Instr. Order

add $1,
stall

sub $4,$1,$5

and $6,$1,$7

Can fix data hazard by waiting – stall – but impacts CPI

Instr. Order

add $1,
stall

sub $4,$1,$5

and $6,$1,$7
Another Way to “Fix” a Data Hazard

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

Instr. Order

add $1,
sub $4,$1,$5
and $6,$1,$7
or $8,$1,$9
xor $4,$1,$5
Data Forwarding (aka Bypassing)

- Take the result from the earliest point that it exists in any of the pipeline state registers and forward it to the functional units (e.g., the ALU) that need it that cycle.

- For ALU functional unit: the inputs can come from any pipeline register rather than just from ID/EX by:
  - adding multiplexors to the inputs of the ALU
  - connecting the Rd write data in EX/MEM or MEM/WB to either (or both) of the EX’s stage Rs and Rt ALU mux inputs
  - adding the proper control hardware to control the new muxes

- Other functional units may need similar forwarding logic (e.g., the DM).

- With forwarding can achieve a CPI of 1 even in the presence of data dependencies.
Data Forwarding Control Conditions

1. EX Forward Unit:
   if (EX/MEM.RegWrite
   and (EX/MEM.RegisterRd != 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
   ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
   ForwardB = 10

1. MEM Forward Unit:
   if (MEM/WB.RegWrite
   and (MEM/WB.RegisterRd != 0)
   and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
   ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
   ForwardB = 01

Forwards the result from the previous instr. to either input of the ALU

Forwards the result from the second previous instr. to either input of the ALU
Forwarding Illustration

Instr. Order

add $1,

sub $4, $1, $5

and $6, $7, $1

EX forwarding  MEM forwarding
Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?

```
Instr. Order
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
```
Corrected Data Forwarding Control Conditions

1. **EX Forward Unit:**
   if (EX/MEM.RegWrite
   and (EX/MEM.RegisterRd != 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
   ForwardA = 10
   if (EX/MEM.RegWrite
   and (EX/MEM.RegisterRd != 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
   ForwardB = 10

1. **MEM Forward Unit:**
   if (MEM/WB.RegWrite
   and (MEM/WB.RegisterRd != 0)
   and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
   and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
   ForwardA = 01
   if (MEM/WB.RegWrite
   and (MEM/WB.RegisterRd != 0)
   and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
   and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
   ForwardB = 01

Forwards the result from the previous instr. to either input of the ALU
Forwards the result from the previous or second previous instr. to either input of the ALU
Datapath with Forwarding Hardware

Instruction Memory
- Read Address
- Read Addr 1
- Read Addr 2
- Read Data 1
- Read Data 2
- Write Addr
- Write Data
- 16
- 32
- Sign Extend

Register
- Read
- Write

File
- Read Addr 1
- Read Addr 2
- Read Data 1
- Read Data 2
- Write Addr
- Write Data

ALU
- ALU cntrl
- Shift left 2
- Add

Memory
- Address
- Read Data
- Write Data

Control
- IF/ID
- Control

Forward Unit
- ID/EX.RegisterRt
- ID/EX.RegisterRs
- MEM/WB.RegisterRd
- EX/MEM.RegisterRd

Branch
- MEM/WB
- IF/ID
- ID/EX
- EX/MEM

PCSrc

IF/ID
- Add
- 4
- PC

ID/EX
- Control

EX/MEM
- Add
- Shift left 2
- ALU
- Control

MEM/WB
- MEM/WB
- Forward Unit
- ID/EX
- IF/ID
- Control
Memory-to-Memory Copies

- For loads immediately followed by stores (memory-to-memory copies) can avoid a stall by adding forwarding hardware from the MEM/WB register to the data memory input.
  - Would need to add a Forward Unit and a mux to the MEM stage
Forwarding with Load-use Data Hazards

lw $1,4($2)
sub $4,$1,$5
and $6,$1,$7
or $8,$1,$9
xor $4,$1,$5
Forwarding with Load-use Data Hazards

Instr. Order

lw $1, 4($2)

sub $4, $1, $5
Forwarding with Load-use Data Hazards

1w $1,4 ($2)

stall

sub $4,$1,$5

and $6,$1,$7

or $8,$1,$9

xor $4,$1,$5

Will still need one stall cycle even with forwarding
Load-use Hazard Detection Unit

- Need a Hazard detection Unit in the ID stage that inserts a stall between the load and its use
  
  1. ID Hazard detection Unit:
     
     ```
     if (ID/EX.MemRead
     and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
     or (ID/EX.RegisterRt = IF/ID.RegisterRt)))
     stall the pipeline
     ```

- The first line tests to see if the instruction now in the EX stage is a `lw`; the next two lines check to see if the destination register of the `lw` matches either source register of the instruction in the ID stage (the load-use instruction)

- After this one cycle stall, the forwarding logic can handle the remaining data hazards
Hazard/Stall Hardware

- Along with the Hazard Unit, we have to implement the stall
- Prevent the instructions in the IF and ID stages from progressing down the pipeline – done by preventing the PC register and the IF/ID pipeline register from changing
  - Hazard detection Unit controls the writing of the PC (PC.write) and IF/ID (IF/ID.write) registers
- Insert a “bubble” between the `lw` instruction (in the EX stage) and the load-use instruction (in the ID stage) (i.e., insert a `noop` in the execution stream)
  - Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 (`noop`). The Hazard Unit controls the mux that chooses between the real control values and the 0’s.
- Let the `lw` instruction and the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline
- A smart compiler will try to avoid load-use stalls by executing instructions which do not require the result of the load in the slot immediately after that load.
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
Datapath Branch and Jump Hardware

Instruction Memory
- Read Address
- Read Addr 1
- Read Addr 2
- Write Addr
- Write Data

Register File
- Read Data 1
- Read Data 2
- Write Data

ALU
- ALU cntrl
- Address
- Read Data
- Write Data

Forward Unit

PC+4[31-28]

Shift left 2

Control

ID/EX

EX/MEM

MEM/WB

Branch

Jump

PCSrc

Data Memory

Forward Unit
Jumps Incur One Stall

- Jumps not decoded until ID, so one flush is needed
  - To flush, set IF.Flush to zero the instruction field of the IF/ID pipeline register (turning it into a noop)

- Unfortunately, jumps are very infrequent – only 3% of the SPECint instruction mix

Diagram:

- Flush
- Jumps not decoded until ID
- IF.Flush
- Speculative decode on every cycle
- Cache bypass to avoid stalls
- If target of jump is speculative, flush
- Speculative decode on every cycle
- Cache bypass to avoid stalls
- If target of jump is speculative, flush
Two “Types” of Stalls

- Noop instruction (or bubble) inserted between two instructions in the pipeline (as done for load-use situations)
  - 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)
  - Insert noop by zeroing control bits in the pipeline register at the appropriate stage
  - Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline

- Flushes (or instruction squashing) where an instruction in the pipeline is replaced with a noop instruction (as done for instructions located sequentially after \( j \) instructions)
  - Zero the control bits for the instruction to be flushed
Supporting ID Stage Jumps

IF/ID
- Sign
- Extend
- Add

ID/EX
- Shift left 2
- Control
- ID/EX
- PC+4[31-28]
- Read Addr 1
- Read Addr 2
- Data 1
- File
- Write Addr
- Read Data 2
- Write Data
- Data
- Memory
- Address
- Read Data
- Write Data
- Branch
- PCSrc
- Forward Unit
- ALU
- cntrl
Review: Branch Instr’s Cause Control Hazards

- Dependencies backward in time cause hazards
One Way to “Fix” a Branch Control Hazard

Instr. Order

beq

flush

flush

flush

target of beq

Inst 3

Fix branch hazard by waiting – flush – but affects CPI
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.

Fix branch hazard by waiting – flush.
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

- Add hardware to compute the branch target address and evaluate the branch decision to the ID stage
  - Reduces the number of stall (flush) cycles to one (as 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 comparing and updating the PC adds a mux, a comparator, and an and gate to the ID timing path

- For deeper pipelines, branch decision points can be even later in the pipeline, incurring more stalls

- As is the case with jumps, an optimizing compiler will try to avoid stalls by filling the slot(s) after a branch with instruction(s) which will not require stalls.
ID Branch Forwarding Issues

- **MEM/WB “forwarding”** is taken care of by the normal RegFile write before read operation.

- Need to forward from the EX/MEM pipeline stage to the ID comparison hardware for cases like:

  ```
  if (IDcontrol.Branch and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
    ForwardC = 1
  if (IDcontrol.Branch and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
    ForwardD = 1
  ```

  Forwards the result from the second previous instr. to either input of the compare.
ID Branch Forwarding Issues, con’t

- If the instruction immediately before the branch produces one of the branch source operands, then a stall needs to be inserted (between the beq and add1) since the EX stage ALU operation is occurring at the same time as the ID stage branch compare operation.
  - “Bounce” the beq (in ID) and next_seq_instr (in IF) in place (ID Hazard Unit deasserts PC.Write and IF/ID.Write)
  - Insert a stall between the add in the EX stage and the beq in the ID stage by zeroing the control bits going into the ID/EX pipeline register (done by the ID Hazard Unit)
- If the branch is found to be taken, then flush the instruction currently in IF (IF.Flush)
- An optimizing compiler should be aware of this problem and avoid it whenever possible.

WB    add3  $3,
MEM   add2  $4,
EX    add1  $1,
ID    beq   $1,$2,Loop
IF    next_seq_instr

Revised sjh 20131124
ID Branch Forwarding Issues with Load Word

- If the instruction immediately before the branch produces is a load of one of the branch source operands, then **two stalls** needs to be inserted (between the \texttt{beq} and \texttt{lw}) since the MEM stage read operation of \texttt{lw} occurs **one clock cycle after** the ID stage branch compare operation.

  - “Bounce” the \texttt{beq} (in ID) and \texttt{next_seqInstr} (in IF) in place for two clock cycles.
  - Insert two stalls between the \texttt{lw} in the EX stage and the \texttt{beq} in the ID stage.

- If the branch is found to be taken, then flush the instruction currently in IF (\texttt{IF.Flush})

- An optimizing compiler should be aware of this problem and avoid it whenever possible.
ID Branch Forwarding Issues with Load Word – 2

- Even if a load of a branch argument occurs two instructions before the branch, a stall needs to be inserted (somewhere between the `beq` and `lw`) since the MEM stage read operation for `lw` is occurring at the same time as the ID stage branch compare operation.
  - “Bounce” the `beq` (in ID) and `next_seq_instr` (in IF) in place for two clock cycles.
  - Insert a stall somewhere between the `lw` in the MEM stage and the `beq` in the ID stage.

- If the branch is found to be taken, then flush the instruction currently in IF (`IF.Flush`)

- An optimizing compiler should be aware of this problem and avoid it whenever possible.
Supporting ID Stage Branches
Delay Slots

- **Observation**: With normal flow of control, for all jumps, as well as for branches which are taken, the next instruction in the pipeline must always be squashed, even with maximum forwarding and dedicated adders.
  - The target address is not known until the jump/branch instruction is decoded.
  - This amounts to “wasted” processor cycles.

- **One Solution** *(delay slot)*: As part of the instruction semantics, require that the instruction immediately following a jump or branch always be executed.
  - The idea is that it may be possible to do something useful in that slot, without knowing the destination of the jump or whether the branch is taken.
  - It is always possible to fill the delay slot with a \texttt{nop} instruction if nothing better can be found.
Delay Slots in MIPS32

- **Standard**: The MIPS32 architecture specifies delay slots for all jump and (ordinary) branch instructions.

- **Consequence**: Some of the code snippets shown in previous slides are not quite correct.

- **Example**:
  ```assembly
  beq  $s0, $s1, Else
  add  $s3, $s0, $s1
  j    Exit
  Else: sub  $s3, $s0, $s1
  Exit: ...
  ```

  should be:
  ```assembly
  beq  $s0, $s1, Else
  nop
  add  $s3, $s0, $s1
  j    Exit
  nop
  Else: sub  $s3, $s0, $s1
  Exit: ...
  ```

  (with nop used as a filler for the delay slots).

- **Further caveat**: MIPS assemblers, as well as SPIM and its relatives, fill delay slots automatically (unless a special flag is set).
  - Thus, the first snippet above would be converted to the second.
Issues in the Use of Delay Slots

- The instructions which are placed in delay slots must be chosen with care:
  - Delay slots may not be filled with other jump or branch instructions.
    - This would lead to nested delay slots – difficult to handle.
  - The instruction in the delay slot for an instance of `jal` should not alter `$ra`.
    - This could lead to a race condition for the value of `$ra`.
  - There are others …

- Special handling is required for instructions in delay slots:
  - Exception handling.

- The details are beyond the scope of this course.
Effective Use of Delay Slots

- The motivation behind introducing delay slot is to try to do something useful in such slots (rather than just a *nop*).
  - In this way, the time wasted on a squashed instruction may be reclaimed to do something useful.
- It is the job of an optimizing compiler to fill delay slots appropriately.
- Some examples will help illustrate the idea.
Scheduling Branch Delay Slots

A. From before branch

```assembly
add $1,$2,$3
if $2=0 then
  delay slot
```

becomes

```assembly
if $2=0 then
  add $1,$2,$3
```

- A is the best choice, fills delay slot and reduces IC

B. From branch target

```assembly
sub $4,$5,$6
add $1,$2,$3
if $1=0 then
  delay slot
```

becomes

```assembly
add $1,$2,$3
if $1=0 then
  sub $4,$5,$6
```

- In B and C, the `sub` instruction may need to be copied, increasing IC

C. From fall through

```assembly
add $1,$2,$3
if $1=0 then
  delay slot
```

becomes

```assembly
add $1,$2,$3
if $1=0 then
  sub $4,$5,$6
```

- In B and C, must be okay to execute `sub` when branch fails
Drawbacks of and Alternatives to Delay Slots

- **Drawback**: As pipelines become longer, the number of delay slots which are required increases.
  - Since delay slots are part of the architecture and not the implementation, specifications which are not compatible for delay slots will not be even source-code compatible (without tweaking for delay slots).
  - **Example**: MIPS-X from the late 1980s specified two delay slots for jumps and branches.

- **Consequence**: Delay slots are rarely introduced in new architectures, and are often considered redundant in existing ones.

- **Alternative to delay slots**: Use *branch prediction* to choose a likely next instruction to execute.
  - **Static branch prediction**: For a given branch, always choose the same target.
  - **Dynamic branch prediction**: Use hardware to make an “intelligent” guess for the next instruction to execute.

- Branch prediction, particularly dynamic branch prediction, has become much more common in recent years.
  - The greatly increased power of hardware has made this a feasible option.

- In the following slides, assume that there are no delay slots.
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):
     - 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
Flush with Misprediction (Predict Not Taken)

- To flush the IF stage instruction, assert $IF.Flush$ to zero the instruction field of the IF/ID pipeline register (transforming it into a $noop$).
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

- Predict not taken doesn’t work well for “bottom of the loop” branching structures
Static Branch Prediction, con’t

- Resolve branch hazards by assuming a given outcome and proceeding

1. **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.
  With more hardware, it is possible to try to predict branch behavior *dynamically* during program execution

1. **Dynamic branch prediction** – predict branches at run-time using *run-time* information
Dynamic Branch Prediction

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the lower bits of the PC, contains bit(s) passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was executed.
  - Prediction bit 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 that 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)
    - A 4096 bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)
The BHT predicts *when* a branch is taken, but does not tell *where* it is taken to!

- A branch target buffer (BTB) in the IF stage caches the branch target address, but we also need to fetch the next sequential instruction. The prediction bit in IF/ID selects which “next” instruction will be loaded into IF/ID at the next clock edge.
  - Would need instruction memory with two read ports

- Or the BTB can cache the branch taken *instruction* (not just its address) while the instruction memory is fetching the next sequential instruction.

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

The BTB may be used for jump instructions as well.
1-bit Prediction Accuracy

- A 1-bit predictor can be incorrect twice when not taken
  - Assume predict_bit = 0 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)
  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)

- For 10 times through the loop, branch taken 9 times:
  - 80% accuracy if predict_bit=0 initially;
  - 90% accuracy if predict_bit=1 intially.
2-bit Predictors

- A 2-bit scheme will give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.
  - ... after the first execution of the loop, if a BHT is used to remember previous history.

right 9 times

BHT also stores the initial FSM state

Loop: 1\(^{st}\) loop instr
2\(^{nd}\) loop instr
  
  last loop instr
  bne $1,$2,Loop
  fall out instr

Predict

Taken

Predict 1

Not taken

Predict 0

Not taken

Predict

Not Taken

Predict

Not Taken

Predict

Not Taken

Predict

Not Taken

Predict

Not Taken

right on 1\(^{st}\) iteration
Effectiveness of Predictors Using Two Bits

- Two-bit predictors are especially effective for nested loops:

```plaintext
for i := 1 to m do
  for j := 1 to n do
    <something>
  end do;
end do;
```

- Exiting the inner loop will cause one misprediction.
- However, this misprediction will not force another misprediction on the next pass through the inner loop, provided that pass does not result in an exit.

- Success rates with initial prediction = branch taken:
  - Inner loop: \( \left( \frac{n-1}{n} \right) \times 100\% \)
  - Outer loop: \( \left( \frac{m-1}{m} \right) \times 100\% \)
- This works even for loops of deeper nesting.
Further Comments: MIPS and Branch Prediction

- For some MIPS32 branch instructions, there are alternatives which do not use delay slots.
- The MIPS32 Instruction Reference specifies alternate *branch likely* instructions.
- Example: `beql` (branch on equal likely)
  - This instruction does not use a delay slot, but rather assumes that the branch is taken.
  - Thus, the instruction after the branch is executed only if the branch condition is false. (Otherwise, it is squashed.)
- This is equivalent to static predict taken.
- There is a warning that these instructions may be dropped from future specifications, and so should not be used.
- Also, there is no (apparent) way to specify that delay slots not be used for jump instructions.
Final Comments: Delay Slots and Branch Prediction

- Delay slots are part of the specification of the *architecture*. 
  - The semantics are the same for all implementations of the architecture.
  - When delay slots are specified, their semantics must be respected.
  - Delay slots may not be ignored, even though they are largely redundant for modern hardware.
  - The final result of a computation may depend upon whether such slots are used.

- Both static and dynamic prediction are part of an *implementation*.
  - The final result of the computation does not depend upon whether or not such prediction is used.
  - Only the efficiency is affected.
  - Dynamic prediction has become widespread with modern, powerful hardware.
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 must 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

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

Stage(s)?
- Arithmetic overflow: EX
- Undefined instruction: ID
- TLB or page fault: IF, MEM
- I/O service request: any
- Hardware malfunction: any

Synchronous?
- Arithmetic overflow: yes
- Undefined instruction: yes
- TLB or page fault: yes
- I/O service request: no
- Hardware malfunction: no

- Beware that multiple exceptions can occur simultaneously in a single clock cycle
Multiple Simultaneous Exceptions

- Hardware sorts the exceptions so that the earliest instruction is the one interrupted first.
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., $8000 0180_{hex}$ for arithmetic overflow)

- A way to flush offending instruction and the ones that follow it
Datapath with Controls for Exceptions

Instruction Memory
Read Address

Data Memory
Read Data Address
Write Data

IF/ID
Add
4

Control
Branch

ID/EX
ID.Flush
0
1
0

EX.MEM
EX.Flush
0

MEM/WB

Forward Unit

Forward Unit

Add
Shift left 2

Sign Extend

ALU cntrl

ALU

Compare

Cause
EPC

Instruction Memory
Read Address

RegFile
Read Addr 1
Read Addr 2
Read Data 1
Read Data 2
Write Addr
Write Data

8000 0180_hex

Forward Unit

Hazard Unit

Data Path with Controls for Exceptions
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