Parallelism in wonderland:
are you ready to see how deep the rabbit hole goes?

Branch Hazards and Static Branch Prediction Techniques

Marco D. Santambrogio: marco.santambrogio@polimi.it
Simone Campanoni: xan@eecs.harvard.edu
Dealing with Branches in the Processor Pipeline
The Problem of Branch Hazards
Branch Prediction Techniques
- Static Branch Prediction
**Conditional Branch Instructions**

- **Conditional Branch Instruction:** the branch is taken only if the condition is satisfied. The branch target address is stored in the Program Counter (PC) instead of the address of the next instruction in the sequential instruction stream.

- Examples of branches for MIPS processor:
  - `beq` (*branch on equal*) and `bne` (*branch on not equal*)
  - `beq $s1, $s2, L1`  # go to L1 if ($s1 == $s2)
  - `bne $s1, $s2, L1`  # go to L1 if ($s1 != $s2)
Execution of conditional branches for 5-stage MIPS pipeline

beq $x,$y,offset

<table>
<thead>
<tr>
<th>Instr. Fetch &amp; PC Increm.</th>
<th>Register Read $x e $y</th>
<th>ALU Op. ($x-$y) &amp; (PC+4+offset)</th>
<th>Write of PC</th>
</tr>
</thead>
</table>

- Instruction fetch and PC increment
- Registers read ($x and $y) from Register File.
- ALU operation to compare registers ($x and $y) to derive Branch Outcome (branch taken or branch not taken).
  - Computation of Branch Target Address (PC+4+offset): the value (PC+4) is added to the least significant 16 bit of the instruction after sign extension
- The result of registers comparison from ALU is used to decide the value to be stored in the PC: (PC+4) or (PC+4+offset).
Execution of conditional branches for 5-stage MIPS pipeline

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>ME</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>Instruction Decode</td>
<td>Execution</td>
<td>Memory Access</td>
<td>Write Back</td>
</tr>
</tbody>
</table>

beq $x, $y, offset

- Instr. Fetch & PC Increm.
- Register Read $x e $y
- ALU Op. ($x-$y) & (PC+4+offset)
- Write of PC

- Branch Outcome and Branch Target Address are ready at the end of the EX stage (3\(^{rd}\) stage)
- Conditional branches are solved when PC is updated at the end of the ME stage (4\(^{th}\) stage)
The Problem of Control Hazards

- **Control hazards**: Attempt to make a decision on the next instruction to fetch before the branch condition is evaluated.
- Control hazards arise from the pipelining of conditional branches and other instructions changing the PC.
- Control hazards reduce the performance from the ideal speedup gained by the pipelining since they can make it necessary to stall the pipeline.
Performance of Branch Schemes

What is the performance impact of conditional branches?

Pipeline Speedup = \frac{\text{Pipeline Depth}}{1 + \text{Pipe Stall Cycles per Instruction due to Branches}}

= \frac{\text{Pipeline Depth}}{1 + \text{Branch Frequency} \times \text{Branch Penalty}}
Branch Hazards

- To feed the pipeline we need to fetch a new instruction at each clock cycle, but the branch decision (to change or not change the PC) is taken during the MEM stage.

- This delay to determine the correct instruction to fetch is called **Control Hazard or Conditional Branch Hazard**

- If a branch changes the PC to its target address, it is a **taken branch**

- If a branch falls through, it is **not taken or untaken**.
The branch instruction may or may not change the PC in MEM stage, but the next 3 instructions are fetched and their execution is started.

If the branch is **not taken**, the pipeline execution is OK

If the branch is **taken**, it is necessary to **flush** the next 3 instructions in the pipeline and fetched the \texttt{lw} instruction at the branch target address (L1)
To stall the pipeline until the branch decision is taken (stalling until resolution) and then fetch the correct instruction flow.

- Without forwarding: for three clock cycles
- With forwarding: for two clock cycles

If the branch is *not taken*, the three cycles penalty is not justified ⇒ throughput reduction.

We can assume the *branch not taken*, and flush the next 3 instructions in the pipeline only if the branch will be taken.
Branch Stalls without Forwarding

```
beq $1, $3, L1
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
```

Diagram:

```
IF   ID   EX   ME   WB
stall  stall  stall
IF   ID   EX   ME   WB
IF   ID   EX   ME   WB
IF   ID   EX   ME   WB
```

Diagram labels:

- **IF**: Instruction Fetch
- **ID**: Instruction Decode
- **EX**: Execution
- **ME**: Memory Access
- **WB**: Write Back
Branch Stalls with Forwarding

beq $1, $3, L1
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
Early Evaluation of the PC

- To improve performance in case of branch hazards, we need to add hardware resources to:
  1. Compare registers to derive branch outcome
  2. Compute branch target address
  3. Update the PC register as soon as possible in the pipeline.

- MIPS processor compares registers, computes branch target address and updates PC during ID stage.
MIPS Processor: Early Evaluation of the PC
Each branch costs **one stall** to fetch the correct instruction flow: (PC+4) or branch target address
Consequence of the early evaluation of the branch decision in ID stage:

- In case of \textit{add} (or \textit{load}) instruction followed by a \textit{branch} testing the result
  \[ \Rightarrow \] we need to introduce \textbf{one stall} before ID stage of branch to enable the forwarding of the result from EX stage of previous instruction

Example:

\begin{verbatim}
addi $1, $1, 4
beq $1, $6, L1
\end{verbatim}
With the branch decision made during ID stage, there is a reduction of the cost associated with each branch (branch penalty):

- We need only one-clock-cycle stall after each branch
- Or a flush of only one instruction following the branch

One-cycle-delay for every branch still yields a performance loss of 10% to 30% depending on the branch frequency:

Pipeline Stall Cycles per Instruction due to Branches = Branch frequency x Branch Penalty

We will examine some techniques to deal with this performance loss.
Branch Prediction Techniques
Branch Prediction Techniques

- In general, the problem of the branch becomes more important for deeply pipelined processors because the cost of incorrect predictions increases (the branches are solved 4 stages or more after the ID stage).

- **Main goal of branch prediction techniques:** try to predict ASAP the result of a branch instruction.

- The performance of a branch prediction technique depends on:
  - **Accuracy** measured in terms of percentage of incorrect predictions.
  - **Cost** of a incorrect prediction measured in terms of time lost to execute useless instructions (misprediction penalty).

- We also need to consider **branch frequency:** the importance of accurate branch prediction is higher in programs with higher branch frequency.
There are many methods to deal with the performance loss due to branch hazards:

- **Static Branch Prediction Techniques:** The actions for a branch are fixed for each branch during the entire execution. The actions are fixed at compile time.

- **Dynamic Branch Prediction Techniques:** The decision causing the branch prediction can change during the program execution.

In both cases, care must be taken not to change the processor state until the branch is definitely known.
Static Branch Prediction Techniques

- Static Branch Prediction is used in processors where the expectation is that the branch behavior is highly predictable at compile time.

- Static Branch Prediction can also be used to assist dynamic predictors.
Static Branch Prediction Techniques

- Branch Always Not Taken (Predicted-Not-Taken)
- Branch Always Taken (Predicted-Taken)
- Backward Taken Forward Not Taken (BTFNT)
- Profile-Driven Prediction
- Delayed Branch
We assume the branch will not taken, thus the sequential instruction flow we have fetched can continue as if the branch condition was not satisfied.

If the condition in stage ID will result not satisfied (the prediction is correct), we can preserve performance.

<table>
<thead>
<tr>
<th>Untaken branch</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>ME</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
</tr>
<tr>
<td>+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
</tr>
</tbody>
</table>
Branch Always Not Taken

- If the condition in stage ID will result satisfied (the prediction is incorrect), the branch is taken:
  - We need to flush the next instruction already fetched (the fetched instruction is turned into a nop) and we restart the execution by fetching the instruction at the branch target address \( \Rightarrow \) One-cycle penalty
An alternative scheme is to consider every branch as taken: as soon as the branch is decoded and the branch target address is computed, we assume the branch to be taken and we begin fetching and executing at the target.

The predicted-taken scheme makes sense for pipelines where the branch target is known before the branch outcome.

In MIPS pipeline, we don’t know the branch target address earlier than the branch outcome, so there is no advantage in this approach for this pipeline.
The prediction is based on the branch direction:

- **Backward-going branches are predicted as taken**
  - Example: the branches at the end of loops go back at the beginning of the next loop iteration

- **Forward-going branches are predicted as not taken**
Profile-Driven Prediction

- The branch prediction is based on profiling information collected from earlier runs.
- The method can use compiler hints.
Delayed Branch Technique

- The compiler statically schedules an independent instruction in the branch delay slot.
- The instruction in the branch delay slot is executed whether or not the branch is taken.
- If we assume a branch delay of one-cycle (as for MIPS) ⇒ we have only one-delay slot
  - Although it is possible to have for some deeply pipeline processors a branch delay longer than one-cycle ⇒ almost all processors with delayed branch have a single delay slot (since it is usually difficult for the compiler to fill in more than one delay slot).
The MIPS compiler always schedules a branch independent instruction after the branch.

Example: A previous `add` instruction without any effects on the branch is scheduled in the Branch Delay Slot.
The behavior of the delayed branch is the same whether or not the branch is taken.

- If the branch is \textit{untaken} $\iff$ execution continues with the instruction after the branch

![Diagram of the delayed branch technique]

- Untaken branch
- Branch delay instr.
- Instr. $i+1$
- Instr. $i+2$
- Instr. $i+3$
**Delayed Branch Technique**

- If the branch is **taken** ⇒ execution continues at the branch target

![Branch diagram]

- Taken branch
- Branch delay instr.
- Branch target instr.
- Branch target instr. + 1
- Branch target instr. + 2
Delayed Branch Technique

- The job of the compiler is to make the instruction placed in the branch delay slot valid and useful.
- There are three ways in which the branch delay slot can be scheduled:
  1. From before
  2. From target
  3. From fall-through
Delayed Branch Technique: From Before

- The branch delay slot is scheduled with an independent instruction from before the branch.
- The instruction in the branch delay slot is **always executed** (whether the branch is taken or untaken).

```
add $1, $2, $3
if $2 == 0 then
  br. delay slot
if $2 == 0 then
  add $1, $2, $3
```
Delayed Branch Technique: From Target

- The use of $1$ in the branch condition prevents \texttt{add} instruction (whose destination is $1$) from being moved after the branch.
- The branch delay slot is scheduled from the target of the branch (usually the target instruction will need to be copied because it can be reached by another path).
- This strategy is preferred when the branch is taken with high probability, such as loop branches (backward branches).

```
sub $4, $5, $6
add $1, $2, $3
if $1 == 0 then
  br. delay slot
sub $4, $5, $6
add $1, $2, $3
if $1 == 0 then
  sub $4, $5, $6
```
Delayed Branch Technique: From Fall-Through

- The use of $1 in the branch condition prevents add instruction (whose destination is $1) from being moved after the branch.
- The branch delay slot is scheduled from the not-taken fall-through path.
- This strategy is preferred when the branch is not taken with high probability, such as forward branches.

```assembly
add $1, $2, $3
if $1 == 0 then
    br. delay slot
or $7, $8, $9
sub $4, $5, $6
```
Delayed Branch Technique

- To make the optimization legal for the target an fall-through cases, it must be OK to execute the moved instruction when the branch goes in the unexpected direction.

- By OK we mean that the instruction in the branch delay slot is executed but the work is wasted (the program will still execute correctly).

- For example, if the destination register is an unused temporary register when the branch goes in the unexpected direction.
In general, the compilers are able to fill about 50% of delayed branch slots with valid and useful instructions, the remaining slots are filled with nops.

In deeply pipeline, the delayed branch is longer than one cycle: many slots must be filled for every branch, thus it is more difficult to fill all the slots with useful instructions.
The main limitations on delayed branch scheduling arise from:

- The restrictions on the instructions that can be scheduled in the delay slot.
- The ability of the compiler to statically predict the outcome of the branch.
Delayed Branch Technique

- To improve the ability of the compiler to fill the branch delay slot ⇒ most processors have introduced a canceling or nullifying branch: the instruction includes the direction that the branch was predicted.
  - When the branch behaves as predicted ⇒ the instruction in the branch delay slot is executed normally.
  - When the branch is incorrectly predicted ⇒ the instruction in the branch delay slot is turned into a NOP
- In this way, the compiler need not be as conservative when filling the delay slot.
Delayed Branch Technique

- MIPS architecture has the **branch-likely** instruction, that behaves as cancel-if-not-taken branch:
  - The instruction in the branch delay slot is executed whether the branch is taken.
  - The instruction in the branch delay slot is **not executed** (it is turned to NOP) whether the branch is untaken.
- Useful approach for backward branches (such as loop branches).
- The branch delay slot must be filled from target.
Questions