Complex Pipelines and Branch Prediction

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
Processor Performance

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}
\]

\[\text{CPI} \quad t_{CK}\]
Processor Performance

\[
\text{Time} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Program}} \cdot \frac{\text{Time}}{\text{Instruction}} \cdot \frac{1}{\text{Cycle}}
\]

\[
\text{CPI} \cdot t_{\text{CK}}
\]

- Pipelining lowers \( t_{\text{CK}} \). What about CPI?
Processor Performance

\[
\text{Time} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}} = CPI \cdot t_{CK}
\]

- Pipelining lowers \( t_{CK} \). What about CPI?

- \( CPI = CPI_{\text{ideal}} + CPI_{\text{hazard}} \)
  - \( CPI_{\text{ideal}} \): cycles per instruction if no stall
Processor Performance

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}
\]

\[
\text{CPI} \cdot t_{CK}
\]

- Pipelining lowers \( t_{CK} \). What about CPI?

- \( \text{CPI} = \text{CPI}_{\text{ideal}} + \text{CPI}_{\text{hazard}} \)
  - \( \text{CPI}_{\text{ideal}} \): cycles per instruction if no stall

- \( \text{CPI}_{\text{hazard}} \) contributors
  - Data hazards: long operations, cache misses
  - Control hazards: branches, jumps, exceptions
5-Stage Pipelined Processors

- Advantages
  - $\text{CPI}_{\text{ideal}} = 1$ (pipelining)
  - Simple and elegant
    - Still used in ARM & MIPS processors
5-Stage Pipelined Processors

- Advantages
  - $CPI_{\text{ideal}} = 1$ (pipelining)
  - Simple and elegant
    - Still used in ARM & MIPS processors

- Shortcomings
  - Upper performance bound is CPI=1
  - High-latency instructions not handled well
    - 1 stage for access to large caches or multiplier
    - Long clock cycle time
  - Unnecessary stalls due to rigid pipeline
    - If one instruction stalls, anything behind stalls
Improving 5-Stage Pipeline Performance

- Increase clock frequency: deeper pipelines
  - Overlap more instructions

- Reduce $\text{CPI}_{\text{ideal}}$: wider pipelines
  - Each pipeline stage processes multiple instructions
Improving 5-Stage Pipeline Performance

- Increase clock frequency: deeper pipelines
  - Overlap more instructions

- Reduce $CPI_{\text{ideal}}$: wider pipelines
  - Each pipeline stage processes multiple instructions

- Reduce impact of data hazards: out-of-order execution
  - Execute each instruction as soon as its source operands are available
Improving 5-Stage Pipeline Performance

- Increase clock frequency: deeper pipelines
  - Overlap more instructions

- Reduce CPI\textsubscript{ideal}: wider pipelines
  - Each pipeline stage processes multiple instructions

- Reduce impact of data hazards: out-of-order execution
  - Execute each instruction as soon as its source operands are available

- Reduce impact of control hazards: branch prediction
  - Predict both target and direction of jumps and branches
Improving 5-Stage Pipeline Performance

- Increase clock frequency: deeper pipelines
  - Overlap more instructions
- Reduce CPI_{ideal}: wider pipelines
  - Each pipeline stage processes multiple instructions
- Reduce impact of data hazards: out-of-order execution
  - Execute each instruction as soon as its source operands are available
- Reduce impact of control hazards: branch prediction
  - Predict both target and direction of jumps and branches

Demystification, not on the Quiz
Deeper Pipelines

- Break up datapath into N pipeline stages
  - Ideal $t_{CK} = 1/N$ compared to non-pipelined
  - So let’s use a large N!
Deeper Pipelines

- Break up datapath into N pipeline stages
  - Ideal $t_{CK} = 1/N$ compared to non-pipelined
  - So let’s use a large N!
Deeper Pipelines

- Break up datapath into $N$ pipeline stages
  - Ideal $t_{\text{CK}} = 1/N$ compared to non-pipelined
  - So let’s use a large $N$!

- Advantage: Higher clock frequency
  - The workhorse behind multi-GHz processors
  - Opteron: 11, Power5: 17, Pentium4: 34; Nehalem: 16
Deeper Pipelines

- Break up datapath into N pipeline stages
  - Ideal $t_{CK} = 1/N$ compared to non-pipelined
  - So let’s use a large N!

- Advantage: Higher clock frequency
  - The workhorse behind multi-GHz processors
  - Opteron: 11, Power5: 17, Pentium4: 34; Nehalem: 16

- Cost
Deeper Pipelines

- Break up datapath into $N$ pipeline stages
  - Ideal $t_{CK} = 1/N$ compared to non-pipelined
  - So let’s use a large $N$!

- Advantage: Higher clock frequency
  - The workhorse behind multi-GHz processors
  - Opteron: 11, Power5: 17, Pentium4: 34; Nehalem: 16

- Cost
  - More pipeline registers
  - Complexity: more bypass paths and control logic
Deeper Pipelines

- Break up datapath into N pipeline stages
  - Ideal $t_{CK} = 1/N$ compared to non-pipelined
  - So let’s use a large N!

- Advantage: Higher clock frequency
  - The workhorse behind multi-GHz processors
  - Opteron: 11, Power5: 17, Pentium4: 34; Nehalem: 16

- Cost
  - More pipeline registers
  - Complexity: more bypass paths and control logic

- Disadvantages
  - More overlapping $\Rightarrow$ more dependencies
    - $CPI_{hazard}$ grows due to data and control hazards
  - Clock overhead becomes increasingly important
  - Power consumption
Wider (aka Superscalar) Pipelines

- Each stage operates on up to $W$ instructions each clock cycle
Wider (aka Superscalar) Pipelines

- Each stage operates on up to $W$ instructions each clock cycle

- Advantage: Lower $\text{CPI}_{\text{ideal}} \left(\frac{1}{W}\right)$
  - Opteron: 3, Pentium4: 3, Nehalem: 4, Power7: 8
Wider (aka Superscalar) Pipelines

- Each stage operates on up to \( W \) instructions each clock cycle

- Advantage: Lower \( \text{CPI}_{\text{ideal}} \) \((1/W)\)
  - Opteron: 3, Pentium4: 3, Nehalem: 4, Power7: 8

- Cost
  - Need wider path to instruction cache
  - Need more ALUs, register file ports, ...
  - Complexity: more bypass & stall cases to check
Wider (aka Superscalar) Pipelines

- Each stage operates on up to $W$ instructions each clock cycle

- Advantage: Lower $\text{CPI}_{\text{ideal}} (1/W)$
  - Opteron: 3, Pentium4: 3, Nehalem: 4, Power7: 8

- Cost
  - Need wider path to instruction cache
  - Need more ALUs, register file ports, ...
  - Complexity: more bypass & stall cases to check

- Disadvantages
  - Parallel execution $\Rightarrow$ more dependencies
    - $\text{CPI}_{\text{hazard}}$ grows due to data and control hazards
Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated

- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value
Resolving Hazards

- **Strategy 1: Stall.** Wait for the result to be available by freezing earlier pipeline stages

- **Strategy 2: Bypass.** Route data to the earlier pipeline stage as soon as it is calculated

- **Strategy 3: Speculate**
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

- **Strategy 4: Find something else to do**
Resolving Hazards

- **Strategy 1: Stall.** Wait for the result to be available by freezing earlier pipeline stages

- **Strategy 2: Bypass.** Route data to the earlier pipeline stage as soon as it is calculated

- **Strategy 3: Speculate**
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

- **Strategy 4: Find something else to do**
Out-of-Order Execution

- Consider the expression \( D = 3(a - b) + 7ac \)
Out-of-Order Execution

- Consider the expression \( D = 3(a - b) + 7ac \)

**Sequential code**

- ld a
- ld b
- sub a-b
- mul 3(a-b)
- ld c
- mul ac
- mul 7ac
- add 3(a-b)+7ac
- st d
Out-of-Order Execution

- Consider the expression \( D = 3(a - b) + 7ac \)

**Sequential code**
- ld a
- ld b
- sub a-b
- mul 3(a-b)
- ld c
- mul ac
- mul 7ac
- add 3(a-b)+7ac
- st d

**Dataflow graph**
- ld b
- ld a
- ld c
- \(-\)
- \(*\)
- \(*\)
- \(+\)
- st d
Out-of-Order Execution

- Consider the expression $D = 3(a - b) + 7ac$

Sequential code

```plaintext
ld a
ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
st d
```

Dataflow graph

Out-of-order execution runs instructions as soon as their inputs become available.

November 27, 2018
Out-of-Order Execution Example

- If `ld b` takes a few cycles (e.g., cache miss), can execute instructions that do not depend on `b`.

**Sequential code**

```
ld a
ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
st d
```
Out-of-Order Execution Example

- If \texttt{ld} b takes a few cycles (e.g., cache miss), can execute instructions that do not depend on b

Sequential code:
- \texttt{ld a}
- \texttt{ld b}
- \texttt{sub a-b}
- \texttt{mul 3(a-b)}
- \texttt{ld c}
- \texttt{mul ac}
- \texttt{mul 7ac}
- \texttt{add 3(a-b)+7ac}
- \texttt{st d}

Dataflow graph:
- \texttt{ld b}
- \texttt{ld a}
- \texttt{ld c}
- \texttt{mul}
- \texttt{mul}
- \texttt{add}
- \texttt{st d}

- Completed
- Executing
- Not ready
A Modern Out-of-Order Superscalar Processor
A Modern Out-of-Order Superscalar Processor

Reconstruct dataflow graph
A Modern Out-of-Order Superscalar Processor

- I-Cache
- Fetch Unit
- Decode/Rename
- Dispatch
- Reservation Stations
- Int
- FP
- L/S
- Reorder Buffer
- Retire
- Write Buffer

In Order

Out Of Order

Reconstruct dataflow graph

Execute each instruction as soon as its source operands are available
A Modern Out-of-Order Superscalar Processor

- I-Cache
- Fetch Unit
- Instruction Buffer
- Decode/Rename
- Dispatch
- Branch Predict

Reconstruct dataflow graph

- Reservation Stations

Execute each instruction as soon as its source operands are available

- Reorder Buffer

Write back results in program order

- Retire
- Write Buffer
- D-Cache

In Order

Out Of Order
A Modern Out-of-Order Superscalar Processor

Reconstruct dataflow graph

Execute each instruction as soon as its source operands are available

Write back results in program order

Why is this needed?
Control Flow Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!
Control Flow Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!

Next fetch started

PC

Fetch

Decode

RegRead

Execute

WriteBack

Branch executed

Next fetch started

Fetch

Decode

RegRead

Execute

WriteBack
Control Flow Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!

```
<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>RegRead</th>
<th>Execute</th>
<th>WriteBack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next fetch started

Loose loop

Branch executed

Next fetch started

PC

Next fetch started
Control Flow Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!

- How much work is lost every time pipeline does not follow correct instruction flow?
Control Flow Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!

- How much work is lost every time pipeline does not follow correct instruction flow?

  Loop length x Pipeline width
Control Flow Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!

- How much work is lost every time pipeline does not follow correct instruction flow?
  
  Loop length x Pipeline width

- One branch every 5-20 instructions... performance impact of mispredictions?
RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump? 
  2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JALR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td></td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JALR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JALR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JALR</td>
<td>After Inst. Decode</td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JALR</td>
<td>After Inst. Decode</td>
<td>After Inst. Execute</td>
</tr>
<tr>
<td>Branches</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JALR</td>
<td>After Inst. Decode</td>
<td>After Inst. Execute</td>
</tr>
<tr>
<td>Branches</td>
<td>After Inst. Execute</td>
<td></td>
</tr>
</tbody>
</table>

November 27, 2018
RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JALR</td>
<td>After Inst. Decode</td>
<td>After Inst. Execute</td>
</tr>
<tr>
<td>Branches</td>
<td>After Inst. Execute</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>
Resolving Hazards

- **Strategy 1:** Stall. Wait for the result to be available by freezing earlier pipeline stages

- **Strategy 2:** Bypass. Route data to the earlier pipeline stage as soon as it is calculated

- **Strategy 3:** Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly $\rightarrow$ do nothing
    - Guessed incorrectly $\rightarrow$ kill & restart with correct value

- **Strategy 4:** Find something else to do
Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated

- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

- Strategy 4: Find something else to do
Resolving Hazards

- **Strategy 1:** Stall. Wait for the result to be available by freezing earlier pipeline stages.

- **Strategy 2:** Bypass. Route data to the earlier pipeline stage as soon as it is calculated.

- **Strategy 3:** Speculate
  - Guess a value and continue executing anyway.
  - When actual value is available, two cases:
    - Guessed correctly \( \rightarrow \) do nothing
    - Guessed incorrectly \( \rightarrow \) kill & restart with correct value

- **Strategy 4:** Find something else to do
Static Branch Prediction

- Probability a branch is taken is ~60-70%, but:

  - Some ISAs attach preferred direction hints to branches, e.g., Motorola MC88110
    - bne0 (*preferred taken*)  beq0 (*not taken*)

  - Achieves ~80% accuracy
Dynamic Branch Prediction
Learning from past behavior

PC → Predictor predict

update

Truth/Feedback

Prediction
Dynamic Branch Prediction
Learning from past behavior

- Temporal correlation
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution
Dynamic Branch Prediction

*Learning from past behavior*

- **Temporal correlation**
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution

- **Spatial correlation**
  - Several branches may resolve in a highly correlated manner (a preferred path of execution)
Predicting the Target Address: Branch Target Buffer (BTB)

- BTB is a cache for targets: Remembers last target PC for taken branches and jumps
  - If hit, use stored target as predicted next PC
  - If miss, use PC+4 as predicted next PC
  - After target is known, update if prediction is wrong

2^k-entry direct-mapped BTB (can also be set-associative)

- PC
- Entry PC
- Valid
- Predicted target PC
- Match
- Valid
- Target
Integrating the BTB in the Pipeline

Predict next PC immediately

PC

Fetch

Decode

RegRead

Execute

WriteBack
Integrating the BTB in the Pipeline

Predict next PC immediately

- Fetch
- Decode
- RegRead
- Execute
- WriteBack
Integrating the BTB in the Pipeline

Predict next PC immediately

- Fetch
- Decode
- RegRead
- Execute
- WriteBack

Tight loop
Integrating the BTB in the Pipeline

Predict next PC immediately

Correct misprediction when the right outcome is known

Tight loop

Correct mispred
BTB Implementation Details

2^k-entry direct-mapped BTB

iMem

pc

k

match

tag(pc_i)  target_i  valid

November 27, 2018

MIT 6.004 Fall 2018
BTB Implementation Details

- Unlike caches, it is fine if the BTB produces an invalid next PC
  - It’s just a prediction!
BTB Implementation Details

- Unlike caches, it is fine if the BTB produces an invalid next PC
  - It’s just a prediction!
- Therefore, BTB area & delay can be reduced by
Unlike caches, it is fine if the BTB produces an invalid next PC
  - It’s just a prediction!

Therefore, BTB area & delay can be reduced by
  - Making tags arbitrarily small (match with a subset of PC bits)
BTB Implementation Details

- Unlike caches, it is fine if the BTB produces an invalid next PC
  - It’s just a prediction!
- Therefore, BTB area & delay can be reduced by
  - Making tags arbitrarily small (match with a subset of PC bits)
  - Storing only a subset of target PC bits (fill missing bits from current PC)
Unlike caches, it is fine if the BTB produces an invalid next PC
  - It’s just a prediction!
Therefore, BTB area & delay can be reduced by
  - Making tags arbitrarily small (match with a subset of PC bits)
  - Storing only a subset of target PC bits (fill missing bits from current PC)
  - Not storing valid bits
Unlike caches, it is fine if the BTB produces an invalid next PC
  - It’s just a prediction!
Therefore, BTB area & delay can be reduced by
  - Making tags arbitrarily small (match with a subset of PC bits)
  - Storing only a subset of target PC bits (fill missing bits from current PC)
  - Not storing valid bits
Even small BTBs are very effective!
interface BTB;
  method Addr predict(Addr pc);
  method Action update(Addr pc, Addr nextPC, Bool taken);
endinterface
BTB Interface

interface BTB;
    method Addr predict(Addr pc);
    method Action update(Addr pc, Addr nextPC, Bool taken);
endinterface

- **predict**: Simple lookup to predict nextPC in Fetch stage
BTB Interface

```plaintext
interface BTB;
    method Addr predict(Addr pc);
    method Action update(Addr pc, Addr nextPC, Bool taken);
endinterface
```

- **predict**: Simple lookup to predict nextPC in Fetch stage
- **update**: On a pc misprediction, if the jump or branch at the pc was taken, then the BTB is updated with the new (pc, nextPC). Otherwise, the pc entry is deleted
BTB Interface

interface BTB;
    method Addr predict(Addr pc);
    method Action update(Addr pc, Addr nextPC, Bool taken);
endinterface

- **predict**: Simple lookup to predict nextPC in Fetch stage
- **update**: On a pc misprediction, if the jump or branch at the pc was taken, then the BTB is updated with the new (pc, nextPC). Otherwise, the pc entry is deleted

A BTB is a good way to improve performance on parts 2 and 3 of the design project
Better Branch Direction Prediction

- Consider the following loop:

```assembly
loop: ...
    addi a1, a1, -1
    bnez a1, loop
```
Better Branch Direction Prediction

Consider the following loop:

```
loop: ...
    addi a1, a1, -1
    bnez a1, loop
```

How many mispredictions does the BTB incur per loop?
Better Branch Direction Prediction

- Consider the following loop:

  ```
  loop: ...
  addi a1, a1, -1
  bnez a1, loop
  ```

- How many mispredictions does the BTB incur per loop?
  - One on loop exit
Better Branch Direction Prediction

- Consider the following loop:

```
loop: ...
    addi a1, a1, -1
    bnez a1, loop
```

- How many mispredictions does the BTB incur per loop?
  - One on loop exit
  - Another one on first iteration
Two-Bit Direction Predictor
Smith 1981

- Use two bits per BTB entry instead of one valid bit
- Manage them as a saturating counter:

<table>
<thead>
<tr>
<th>On not-taken</th>
<th>On taken</th>
<th>Direction Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Strongly taken</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Weakly taken</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Weakly not-taken</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Strongly not-taken</td>
</tr>
</tbody>
</table>

- Direction prediction changes only after two wrong predictions
Two-Bit Direction Predictor
Smith 1981

- Use two bits per BTB entry instead of one valid bit
- Manage them as a saturating counter:

<table>
<thead>
<tr>
<th></th>
<th>On not-taken</th>
<th>On taken</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Strongly taken</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Weakly taken</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Weakly not-taken</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Strongly not-taken</td>
</tr>
</tbody>
</table>

- Direction prediction changes only after two wrong predictions

- How many mispredictions per loop?
Two-Bit Direction Predictor
Smith 1981

- Use two bits per BTB entry instead of one valid bit
- Manage them as a saturating counter:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>On not-taken</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>On taken</strong></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Strongly taken</strong></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Weakly taken</strong></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Weakly not-taken</strong></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Strongly not-taken</strong></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Direction prediction changes only after two wrong predictions

- How many mispredictions per loop? 1
Modern Processors Combine Multiple Specialized Predictors

- Predict next PC immediately
  - Instruction type & branch/JAL target known
  - Branch direction & JALR target known

Fetch

Decode

RegRead

Execute

WriteBack

BTB

Branch dir predictor

Return addr predictor

Loop predictor

Correct mispred
Modern Processors Combine Multiple Specialized Predictors

- **Fetch**
  - Predict next PC immediately
  - Instruction type & branch/JAL target known

- **RegRead**
  - Branch direction & JALR target known

- **Execute**
  - Correct mispred

- **WriteBack**

Best predictors reflect program behavior

Branch dir predictor
Return addr predictor
Loop predictor
Summary

- Modern processors rely on a handful of techniques
  - Deep pipelines → Multi-GHz frequency
  - Wide (superscalar) pipelines → Multiple instructions/cycle
  - Out-of-order execution → Reduce impact of data hazards
  - Branch prediction → Reduce impact of control hazards
Summary

- Modern processors rely on a handful of techniques
  - Deep pipelines → Multi-GHz frequency
  - Wide (superscalar) pipelines → Multiple instructions/cycle
  - Out-of-order execution → Reduce impact of data hazards
  - Branch prediction → Reduce impact of control hazards

- Main objective is high sequential performance
  - High costs (area, power)
  - Requires high memory bandwidth and low latency
  - Simple to use (but knowing these techniques often needed to write high-performance programs!)
Thank you!

Next lecture: Bypassing in Bluespec
+ Design Project Tips