Branch Prediction

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Reminder: Phases of Instruction Execution

In order

- Fetch: Instruction bits retrieved from cache.
- Decode: Instructions placed in appropriate issue (aka “dispatch”) stage buffer
- Execute: Instructions and operands sent to execution units. When execution completes, all results and exception flags are available.
- Commit: Instruction irrevocably updates architectural state (aka “graduation” or “completion”).

Out of order

- Results Buffer
- Func. Units
- Issue Buffer
- Fetch Buffer
- I-cache
- PC
Control Flow Penalty

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

How much work is lost if pipeline doesn’t follow correct instruction flow?
Control Flow Penalty

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

How much work is lost if pipeline doesn’t follow correct instruction flow?

~ Loop length x pipeline width
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!

How much work is lost if pipeline doesn’t follow correct instruction flow?

≈ Loop length x pipeline width
Average Run-Length between Branches

Average dynamic instruction mix of SPEC CPU 2017
[Limaye and Adegbiya, ISPASS’18]:

<table>
<thead>
<tr>
<th></th>
<th>SPECInt</th>
<th>SPECfp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branches</td>
<td>19 %</td>
<td>11 %</td>
</tr>
<tr>
<td>Loads</td>
<td>24 %</td>
<td>26 %</td>
</tr>
<tr>
<td>Stores</td>
<td>10 %</td>
<td>7 %</td>
</tr>
<tr>
<td>Other</td>
<td>47 %</td>
<td>56 %</td>
</tr>
</tbody>
</table>

SPECInt17: perlbench, gcc, mcf, omnetpp, xalancbmk, x264, 
deepsjeng, leela, exchange2, xz
SPECfp17: bwaves, cactus, lbm, wrf, pop2, imagick, nab, fotonik3d, roms

What is the average run length between branches?
Average Run-Length between Branches

Average dynamic instruction mix of SPEC CPU 2017
[Limaye and Adegbiya, ISPASS’18]:

<table>
<thead>
<tr>
<th></th>
<th>SPECInt</th>
<th>SPECfp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branches</td>
<td>19 %</td>
<td>11 %</td>
</tr>
<tr>
<td>Loads</td>
<td>24 %</td>
<td>26 %</td>
</tr>
<tr>
<td>Stores</td>
<td>10 %</td>
<td>7 %</td>
</tr>
<tr>
<td>Other</td>
<td>47 %</td>
<td>56 %</td>
</tr>
</tbody>
</table>

SPECInt17: perlbench, gcc, mcf, omnetpp, xalancbmk, x264, deepsjeng, leela, exchange2, xz
SPECfp17: bwaves, cactus, lbm, wrf, pop2, imagick, nab, fotonik3d, roms

What is the average run length between branches?

Roughly 5-10 instructions
RISC-V Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
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>BRANCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e.g., BLT)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RISC-V Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
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>BRANCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e.g., BLT)</td>
<td>After Inst. Decode</td>
<td></td>
</tr>
</tbody>
</table>
# RISC-V Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
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>BRANCH (e.g., BLT)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# RISC-V Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?

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. Decode</td>
</tr>
<tr>
<td>BRANCH (e.g., BLT)</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>
RISC-V Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
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 Reg. Fetch</td>
</tr>
<tr>
<td>BRANCH (e.g., BLT)</td>
<td>After Inst. Decode</td>
<td></td>
</tr>
</tbody>
</table>

# RISC-V Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
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 Reg. Fetch</td>
</tr>
<tr>
<td>BRANCH (e.g., BLT)</td>
<td>After Inst. Execute</td>
<td></td>
</tr>
</tbody>
</table>
RISC-V Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
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 Reg. Fetch</td>
</tr>
<tr>
<td>BRANCH (e.g., BLT)</td>
<td>After Inst. Execute</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>
Example Branch Penalties

UltraSPARC-III instruction fetch pipeline stages
(in-order issue, 4-way superscalar, 750MHz, 2000)

A  PC Generation/Mux
P  Instruction Fetch Stage 1
F  Instruction Fetch Stage 2
B  Branch Address Calc/Begin Decode
I  Complete Decode
J  Steer Instructions to Functional units
R  Register File Read
E  Integer Execute

Remainder of execute pipeline
(+ another 6 stages)
Example Branch Penalties

UltraSPARC-III instruction fetch pipeline stages
(in-order issue, 4-way superscalar, 750MHz, 2000)

Branch
Target
Address
Known

A  PC Generation/Mux
P  Instruction Fetch Stage 1
F  Instruction Fetch Stage 2
B  Branch Address Calc/Begin Decode
I  Complete Decode
J  Steer Instructions to Functional units
R  Register File Read
E  Integer Execute

Remainder of execute pipeline
(+ another 6 stages)
Example Branch Penalties

UltraSPARC-III instruction fetch pipeline stages (in-order issue, 4-way superscalar, 750MHz, 2000)

Branch
Target
Address
Known

Jump
Register
Target
Known

A
PC Generation/Mux

P
Instruction Fetch Stage 1

F
Instruction Fetch Stage 2

B
Branch Address Calc/Begin Decode

I
Complete Decode

J
Steer Instructions to Functional units

R
Register File Read

E
Integer Execute

Remainder of execute pipeline (+ another 6 stages)
Example Branch Penalties

UltraSPARC-III instruction fetch pipeline stages
(in-order issue, 4-way superscalar, 750MHz, 2000)

PC Generation/Mux
Instruction Fetch Stage 1
Instruction Fetch Stage 2
Branch Address Calc/Begin Decode
Complete Decode
Steer Instructions to Functional units
Register File Read
Integer Execute
Remainder of execute pipeline (+ another 6 stages)

Branch
Target
Address
Known
Jump
Register
Target
Known
Branch
Direction
Known
Reducing Control Flow Penalty

- Software solutions

- Hardware solutions
Reducing Control Flow Penalty

• Software solutions
  – *Eliminate branches – loop unrolling*
    Increases run length between branches

• Hardware solutions
Reducing Control Flow Penalty

• Software solutions
  – *Eliminate branches – loop unrolling*
    Increases run length between branches
  – *Reduce resolution time – instruction scheduling*
    Compute the branch condition as early as possible
    (of limited value)

• Hardware solutions
Reducing Control Flow Penalty

• Software solutions
  – *Eliminate branches – loop unrolling*
    Increases run length between branches
  – *Reduce resolution time – instruction scheduling*
    Compute the branch condition as early as possible
    (of limited value)

• Hardware solutions
  – Bypass – usually results are used immediately
Reducing Control Flow Penalty

• Software solutions
  – *Eliminate branches – loop unrolling*
    Increases run length between branches
  – *Reduce resolution time – instruction scheduling*
    Compute the branch condition as early as possible
     (of limited value)

• Hardware solutions
  – Bypass – usually results are used immediately
  – Change architecture – find something else to do
    *Delay slots* – replace pipeline bubbles with useful work
     (requires software cooperation)
Reducing Control Flow Penalty

• Software solutions
  – *Eliminate branches – loop unrolling*
    Increases run length between branches
  – *Reduce resolution time – instruction scheduling*
    Compute the branch condition as early as possible
    (of limited value)

• Hardware solutions
  – Bypass – usually results are used immediately
  – Change architecture – find something else to do
    *Delay slots* – replace pipeline bubbles with useful work
    (requires software cooperation)
  – *Speculate (accurately) – branch prediction*
    Speculative execution of instructions beyond the branch
Branch Prediction

**Motivation:**

Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

**Required hardware support:**

*Prediction structures:*

- Branch history tables, branch target buffers, etc.

*Mispredict recovery mechanisms:*

- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to state following branch
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

- **backward** 90%
- **forward** 50%
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

\textit{bne0} \textit{(preferred taken)} \hspace{1cm} \textit{beq0} \textit{(not taken)}
Static Branch Prediction

Overall probability a branch is taken is \(~60\text{-}70\%\) but:

ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
\hspace{2em} bne0 \((preferred \ taken)\) beq0 \((not \ taken)\)

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64
\hspace{2em} typically reported as \(~80\%\) accurate
Dynamic Prediction
Dynamic Prediction
Dynamic Prediction
Dynamic Prediction

Input → Predictor → Prediction

Truth/Feedback

Operations
- Predict
- Update
Dynamic Prediction

Prediction as a feedback control process

Operations
- Predict
- Update
Dynamic Branch Prediction
Learning based on 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)
Predictor Primitive
Emer & Gloy, 1997

- Indexed table holding values

- Operations
  - Predict
  - Update

- Algebraic notation

Prediction = P[Width, Depth](Index; Update)
One-bit Predictor
aka Branch History Table (BHT)

Simple temporal prediction
1 bit

PC

Prediction

Taken
One-bit Predictor
aka Branch History Table (BHT)

Simple temporal prediction
1 bit

PC

\[ A21064(\text{PC}; \ T) = P[1, 2K](\text{PC}; \ T) \]
One-bit Predictor
aka Branch History Table (BHT)

Simple temporal prediction

1 bit

PC

Prediction

Taken

$A_{21064}(PC; T) = P[1, 2K](PC; T)$

What happens on loop branches?
One-bit Predictor
aka Branch History Table (BHT)

Simple temporal prediction
1 bit

PC

\[ A21064(\text{PC}; T) = P[1, 2K](\text{PC}; T) \]

What happens on loop branches?
At best, mispredicts twice for every use of loop
Two-bit Predictor
Smith, 1981

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

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

– Direction prediction changes only after two wrong predictions
Two-bit Predictor

Smith, 1981

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

<table>
<thead>
<tr>
<th>On taken</th>
<th>On not-taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>Strongly taken</td>
</tr>
<tr>
<td>1 0</td>
<td>Weakly taken</td>
</tr>
<tr>
<td>0 1</td>
<td>Weakly not-taken</td>
</tr>
<tr>
<td>0 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 Predictor
Smith, 1981

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

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

- Direction prediction changes only after two wrong predictions

How many mispredictions per loop? 1
Two-bit Predictor

Smith, 1981
Two-bit Predictor

Smith, 1981

Counter\[W,D\](I; T) = P\[W, D\](I; if T then P+1 else P-1)

A21164(PC; T) = MSB(Counter[2, 2K](PC; T))
<table>
<thead>
<tr>
<th>Branch History Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fetch PC</strong></td>
</tr>
<tr>
<td>00</td>
</tr>
</tbody>
</table>
Branch History Table

Fetch PC

I-Cache

Instruction

Opcode  offset
Branch History Table

Fetch PC

I-Cache

Instruction

 Opcode

 offset

Branch?

Target PC

+ 010
Branch History Table

Fetch PC

I-Cache

Instruction

Opcode

offset

Branch?

Target PC

BHT Index

$2^k$-entry BHT, 2 bits/entry

Taken/¬Taken?
Branch History Table

4K-entry BHT, 2 bits/entry, ~80-90% correct predictions
Exploiting Spatial Correlation
Yeh and Patt, 1992

if (x[i] < 7) then
  y += 1;
if (x[i] < 5) then
  c -= 4;

If first condition false, second condition also false
Exploiting Spatial Correlation
Yeh and Patt, 1992

if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;

If first condition false, second condition also false

*History register* records the direction of the last N branches executed by the processor
History Registers
aka Pattern History Table (PHT)

PC

I

P

U

Concatenate

History

Taken
History Registers
aka Pattern History Table (PHT)

\[
\text{History}(PC; T) = P(PC; P \ || \ T)
\]
Global-History Predictor

0

Global History

Concat

Prediction

+/-

Taken
Global-History Predictor

\[
G\text{Hist}(;T) = \text{MSB}(\text{Counter}(\text{History}(0, T); T))
\]
Global-History Predictor

\[ \text{GHist}(;T) = \text{MSB}(\text{Counter}(\text{History}(0, T); T)) \]

Can we take advantage of a pattern at a particular PC?
Local-History Predictor
Local-History Predictor

\[ \text{LHist}(PC; T) = \text{MSB}(\text{Counter}(\text{History}(PC; T); T)) \]
Local-History Predictor

Can we take advantage of the global pattern at a particular PC?

LHist(PC; T) = MSB(Counter(History(PC; T); T))
Global-History Predictor with Per-PC Counters

GHistPA(PC; T) = MSB(Counter(History(0; T) || PC; T))
Global-History Predictor with Per-PC Counters

\[
G\text{HistPA}(PC; T) = \text{MSB}(\text{Counter}(\text{History}(0; T) || PC; T))
\]

\[
G\text{Share}(PC; T) = \text{MSB}(\text{Counter}(\text{History}(0; T) \oplus PC; T))
\]
Two-Level Branch Predictor
(Pentium Pro, 1995)

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)

- Fetch PC
- 2-bit global branch history shift register
- Shift in Taken/¬Taken results of each branch
- Taken/¬Taken?
Choosing Predictors

LHist → Prediction

GHist → Prediction

Chooser → Prediction
Choosing Predictors

\[ \text{Chooser} = \text{MSB}(P(\text{PC}; P + (A==T) - (B==T))) \]

or

\[ \text{Chooser} = \text{MSB}(P(\text{GHist}(\text{PC}; T); P + (A==T) - (B==T))) \]
Tournament Branch Predictor
(Alpha 21264, 1996)

- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications
TAGE predictor
Seznec & Michaud, 2006

PC

Bimodal

TAGE[L1]

TAGE[L2]

TAGE[L3]

My guess

Use me?

Final Prediction
TAGE predictor
Seznec & Michaud, 2006

TAGE_TREE[L1, L2, L3](PC; T) =
TAGE[L3](PC, TAGE[L2](PC, TAGE[L1](PC, Bimodal(PC;T)
;T) ;T) ;T)

My guess
Use me?

Final Prediction
TAGE component

Next Predictor

GHist

PC

Counter

Useful

Tag

My guess

Prediction

Use me?
TAGE predictor component
TAGE predictor component

\[
\text{TAGE}[L](PC, \text{NEXT}; T) =
\]
\[
\begin{align*}
\text{idx} &= \text{hash}(PC, \text{GHIST}[L](;T)) \\
\text{tag} &= \text{hash'}(PC, \text{GHIST}[L](;T)) \\
\text{TAGE.U} &= \text{SA}(\text{idx, tag}; ((\text{TAGE} == T) && (\text{NEXT} != T))?1:\text{SA}) \\
\text{TAGE.Counter} &= \text{SA}(\text{idx, tag}; T?\text{SA}+1:\text{SA}-1)
\end{align*}
\]
\[
\text{use_me} = \text{TAGE.U} && \text{isStrong} (\text{TAGE.Counter})
\]
\[
\text{TAGE} = \text{use_me}?\text{MSB} (\text{TAGE.Counter}):\text{NEXT}
\]

Notes:

- \text{SA} is a set-associative structure
- \text{SA} allocation occurs on mispredict (not shown)
- \text{TAGE.U} cleared on global counter saturation
Limitations of branch predictors

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.

<table>
<thead>
<tr>
<th>A</th>
<th>PC Generation/Mux</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>F</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>B</td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td>I</td>
<td>Complete Decode</td>
</tr>
<tr>
<td>J</td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td>R</td>
<td>Register File Read</td>
</tr>
<tr>
<td>E</td>
<td>Integer Execute</td>
</tr>
</tbody>
</table>

Remainder of execute pipeline (+ another 6 stages)

*UltraSPARC-III fetch pipeline*
Limitations of branch predictors

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.

Correctly predicted taken branch penalty

UltraSPARC-III fetch pipeline

A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute

Remainder of execute pipeline (+ another 6 stages)
Limitations of branch predictors

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.

UltraSPARC-III fetch pipeline

Correctly predicted taken branch penalty

Jump Register penalty

A  PC Generation/Mux
P  Instruction Fetch Stage 1
F  Instruction Fetch Stage 2
B  Branch Address Calc/Begin Decode
I  Complete Decode
J  Steer Instructions to Functional units
R  Register File Read
E  Integer Execute

Remainder of execute pipeline (+ another 6 stages)
Branch Target Buffer (untagged)

BP bits are stored with the predicted target address.

IF stage: If \((BP=\text{taken})\) then \(nPc=\text{target}\) else \(nPc=\text{PC}+4\)

later: \(\text{check prediction, if wrong then kill the instruction} \quad \text{and update BTB} \quad \text{and BPb, else update BPb}\)
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?
BTB prediction =
Correct target =

⇒

Jump 100
Add ....

Instruction Memory

132
1028
236
take/target

Addition of 132 to 236 yields 155, which is not in the BTB. Thus, a software-based branch prediction is required. If the instruction at 236 is a branch, the correct target is 1028. Otherwise, the target is 100.
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?

BTB prediction = 236
Correct target =

⇒
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?
BTB prediction = 236
Correct target = 1032

⇒
What will be fetched after the instruction at 1028?

BTB prediction = \(236\)
Correct target = \(1032\)

⇒ *kill* PC=236 and *fetch* PC=1032
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?

- BTB prediction = 236
- Correct target = 1032

⇒ **kill** PC=236 and **fetch** PC=1032

*Is this a common occurrence?*
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?

BTB prediction = 236
Correct target = 1032

⇒ kill PC=236 and fetch PC=1032

Is this a common occurrence? Yes
What will be fetched after the instruction at 1028?

- BTB prediction = 236
- Correct target = 1032

⇒ *kill* PC=236 and *fetch* PC=1032

*Is this a common occurrence?*  Yes

*Can we avoid these mispredictions?*
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?

BTB prediction = 236
Correct target = 1032

⇒ *kill* PC=236 and *fetch* PC=1032

*Is this a common occurrence?* Yes
*Can we avoid these mispredictions?* Yes
BTB is only for Control Instructions

BTB contains useful information for branch and jump instructions only
⇒ Do not update it for other instructions

For all other instructions the next PC is (PC)+4!

How to achieve this effect without decoding the instruction?
BTB is only for Control Instructions

BTB contains useful information for branch and jump instructions only

⇒ Do not update it for other instructions

For all other instructions the next PC is (PC)+4!

How to achieve this effect without decoding the instruction?

Tag the entries in the table
- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only \textit{taken} branches and jumps held in BTB
- Next PC determined \textit{before} branch fetched and decoded
Consulting BTB Before Decoding

- The match for PC=1028 fails and 1028+4 is fetched
  ⇒ eliminates false predictions after ALU instructions

- BTB contains entries only for control transfer instructions
  ⇒ more room to store branch targets
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JALR)
- BHT can hold many more entries and is more accurate
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JALR)
- BHT can hold many more entries and is more accurate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC Generation/Mux</td>
<td>Branch Address Calc/Begin Decode</td>
<td>Complete Decode</td>
<td>Steer Instructions to Functional units</td>
<td>Register File Read</td>
<td>Integer Execute</td>
<td>Instruction Fetch Stage 1</td>
</tr>
</tbody>
</table>
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JALR)
- BHT can hold many more entries and is more accurate

PC Generation/Mux
Instruction Fetch Stage 1
Instruction Fetch Stage 2
Branch Address Calc/Begin Decode
Complete Decode
Steer Instructions to Functional units
Register File Read
Integer Execute
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JALR).
- BHT can hold many more entries and is more accurate.

BHT in later pipeline stage corrects when BTB misses a predicted taken branch.
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JALR).
- BHT can hold many more entries and is more accurate.

BTB/BHT only updated after branch resolves in E stage.
Uses of Jump Register (JALR)

- Switch statements (jump to address of matching case)

- Dynamic function call (jump to run-time function address)

- Subroutine returns (jump to return address)
Uses of Jump Register (JALR)

- Switch statements (jump to address of matching case)

- Dynamic function call (jump to run-time function address)

- Subroutine returns (jump to return address)

How well does BTB work for each of these cases?
Uses of Jump Register (JALR)

• Switch statements (jump to address of matching case)
  
  BTB works well if same case used repeatedly

• Dynamic function call (jump to run-time function address)

• Subroutine returns (jump to return address)

How well does BTB work for each of these cases?
Uses of Jump Register (JALR)

- Switch statements (jump to address of matching case)
  
  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)

  BTB works well if same function usually called, (e.g., in C++ programs, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

How well does BTB work for each of these cases?
Uses of Jump Register (JALR)

- Switch statements (jump to address of matching case)
  
  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)
  
  BTB works well if same function usually called, (e.g., in C++ programs, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)
  
  BTB works well if usually return to the same place

How well does BTB work for each of these cases?
Uses of Jump Register (JALR)

• Switch statements (jump to address of matching case)
  
  BTB works well if same case used repeatedly

• Dynamic function call (jump to run-time function address)
  
  BTB works well if same function usually called, (e.g., in C++ programs, when objects have same type in virtual function call)

• Subroutine returns (jump to return address)
  
  BTB works well if usually return to the same place
  
  \[ \Rightarrow \text{Often one function called from many distinct call sites!} \]

How well does BTB work for each of these cases?
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```c
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```java
fa() { fb(); }

fb() { fc(); }

fc() { fd(); }
```

$k$ entries

(typically $k=8$-16)
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```plaintext
fa() { fb(); }
fb() { fc(); }
fcc() { fd(); }
```

**Push call address when function call executed**

- **k entries**
- (typically k=8-16)
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```cpp
fa() { fb(); }
fb() { fc(); }
fcc() { fd(); }
```

*Push call address when function call executed*

- k entries (typically k=8-16)
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```c
fa() { fb(); }

fb() { fc(); }

fc() { fd(); }
```

Push call address when function call executed

![Diagram showing the subroutine return stack with k entries (typically k=8-16)]
Subroutine Return Stack

Small structure to accelerate JR for subroutine
returns, typically much more accurate than BTBs.

```c
fa() { fb(); }

fb() { fc(); }

fc() { fd(); }
```

*Push call address when function call executed*

```
&fd()
&fc()
&fb()
```

$k$ entries

*(typically $k=8-16$)*
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```cpp
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```

Push call address when function call executed

Pop return address when subroutine return decoded

k entries (typically k=8-16)
Line Prediction
(Alpha 21[234]64)

- For superscalar, useful to predict next cache line(s) to fetch
  - Line Predictor predicts line to fetch each cycle (tight loop)
    - Untagged BTB structure – Why?
    - 21464 was to predict 2 lines per cycle
  - Icache fetches block, and predictors improve target prediction
  - PC Calc checks accuracy of line prediction(s)
Overview of Branch Prediction
Overview of Branch Prediction
Overview of Branch Prediction
Overview of Branch Prediction

```
  PC  ----->  Decode  ----->  Reg Read  ----->  Execute
```

Overview of Branch Prediction

- PC
- Decode: Instr type, PC relative targets available
- Reg Read
- Execute
Overview of Branch Prediction

PC → Decode → Reg Read → Execute

Instr type, PC relative targets available
Overview of Branch Prediction

- **PC**
- **Decode**
  - Instr type, PC relative targets available
- **Reg Read**
  - Simple conditions, register targets available
- **Execute**
Overview of Branch Prediction

- Decode: Instr type, PC relative targets available
- Reg Read: Simple conditions, register targets available
- Execute

PC
Overview of Branch Prediction

- Decode: Instr type, PC relative targets available
- Reg Read: Simple conditions, register targets available
- Execute: Complex conditions and exceptions available
Overview of Branch Prediction

- **PC**: Instruction type, PC relative targets available
- **Decode**: Simple conditions, register targets available
- **Reg Read**: Complex conditions and exceptions available
- **Execute**:
Overview of Branch Prediction

P C
Need next PC immediately

... Decode ... Instr type, PC relative targets available

... Reg Read ... Simple conditions, register targets available

... Execute ... Complex conditions and exceptions available
Overview of Branch Prediction

- Need next PC immediately
  - Decode: Instr type, PC relative targets available
  - Reg Read: Simple conditions, register targets available
  - Execute: Complex conditions and exceptions available
Overview of Branch Prediction

Tight loop

Need next PC immediately

BTB

PC

Instr type, PC relative targets available

Decode

Reg Read

Simple conditions, register targets available

Execute

Complex conditions and exceptions available
Overview of Branch Prediction

- Need next PC immediately
  - Tight loop

- Instr type, PC relative targets available
  - Loose loop

- Simple conditions, register targets available
- Complex conditions and exceptions available

BTB

PC

Decode
Reg Read
Execute
Overview of Branch Prediction

- Need next PC immediately
  - Tight loop
- BTB
- Decode
  - Instr type, PC relative targets available
  - Loose loop
- Reg Read
  - Simple conditions, register targets available
  - Loose loop
- Execute
  - Complex conditions and exceptions available
  - Loose loop
Overview of Branch Prediction

Must speculation check always be correct?
Overview of Branch Prediction

Must speculation check always be correct? No...
Overview of Branch Prediction

Must speculation check always be correct?  No...

- **Need next PC immediately**
  - Tight loop
- **Instr type, PC relative targets available**
  - Loose loop
- **Simple conditions, register targets available**
  - Loose loop
- **Complex conditions and exceptions available**
  - Loose loop
Overview of Branch Prediction

- **PC**: Need next PC immediately
- **BTB**: Tight loop
- **Decode**: Loose loop
- **Reg Read**: Loose loop
- **Execute**: Loose loop

Best predictors reflect program behavior

Must speculation check always be correct? No...
Next Lecture:
Speculative Execution
& Value Management