Modern Processor Architecture
Lecture Goals

- Learn about the key techniques that modern processors use to achieve high performance
Lecture Goals

- Learn about the key techniques that modern processors use to achieve high performance

- Emphasize the techniques that may help you in the design project (e.g., increasing pipeline stages, simple branch prediction)
Reminder: Processor Performance

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

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

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

\[\text{CPI} = \frac{t_{\text{CK}}}{t_{\text{CK}}\text{CPI}}\]

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

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

\[\text{CPI} = \frac{\text{t}_{\text{CK}}}{\text{CPI}_{\text{ideal}}} + \text{CPI}_{\text{hazard}}\]

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

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

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

\[ \text{CPI} = \text{CPI}_{\text{ideal}} + \text{CPI}_{\text{hazard}} \]

- \text{CPI}_{\text{ideal}}: cycles per instruction if no stall

\text{CPI}_{\text{hazard}} \text{ contributors}

- Data hazards: long operations, cache misses
- Control hazards: branches, jumps, exceptions
Standard 5-Stage Pipeline

- Assume full bypassing
Standard 5-Stage Pipeline

- Assume full bypassing
- $\text{CPI_{ideal}} = 1.0$
Standard 5-Stage Pipeline

- Assume full bypassing
- $CPI_{\text{ideal}} = 1.0$
Assume full bypassing

$\text{CPI}_{\text{ideal}} = 1.0$

$\text{CPI}_{\text{hazard}}$ due to data hazards: 
*Up to how many cycles lost to each load-to-use hazard?*
Standard 5-Stage Pipeline

- Assume full bypassing
- \( \text{CPI}_{\text{ideal}} = 1.0 \)
- \( \text{CPI}_{\text{hazard}} \) due to data hazards: 
  *Up to how many cycles lost to each load-to-use hazard? 2*
Standard 5-Stage Pipeline

- Assume full bypassing
- \( \text{CPI}_{\text{ideal}} = 1.0 \)
- \( \text{CPI}_{\text{hazard}} \) due to data hazards: *Up to how many cycles lost to each load-to-use hazard?* 2
- \( \text{CPI}_{\text{hazard}} \) due to control hazards: *How many cycles lost to each jump and taken branch?* _____
Standard 5-Stage Pipeline

- Assume full bypassing

- $CPI_{\text{ideal}} = 1.0$

- $CPI_{\text{hazard due to data hazards}}$: Up to how many cycles lost to each load-to-use hazard? 2

- $CPI_{\text{hazard due to control hazards}}$: How many cycles lost to each jump and taken branch? 2
Design Project Pipeline (Part 2)

- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle
Design Project Pipeline (Part 2)

- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle
- \( \text{CPI}_{\text{hazard}} \) due to data hazards: *Up to how many cycles lost to each load-to-use hazard? _____*
Design Project Pipeline (Part 2)

- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle

- $CPI_{\text{hazard}}$ due to data hazards: *Up to how many cycles lost to each load-to-use hazard? _____*
Design Project Pipeline (Part 2)

- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle
- $\text{CPI}_{\text{hazard}}$ due to data hazards: *Up to how many cycles lost to each load-to-use hazard? 1*
Design Project Pipeline (Part 2)

- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle

- \( \text{CPI}_{\text{hazard}} \) due to data hazards: *Up to how many cycles lost to each load-to-use hazard?* \( 1 \)

- \( \text{CPI}_{\text{hazard}} \) due to control hazards: *How many cycles lost to each jump and taken branch?*
Design Project Pipeline (Part 2)

- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle

- $CPI_{\text{hazard}}$ due to data hazards: *Up to how many cycles lost to each load-to-use hazard?* 1

- $CPI_{\text{hazard}}$ due to control hazards: *How many cycles lost to each jump and taken branch?* 1

December 3, 2019
Design Project Pipeline (Part 2)

- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses PC bypassing: On annulment, IF starts fetching at the jump/branch target on the same cycle

- $CPI_{\text{hazard}}$ due to data hazards: 
  \textit{Up to how many cycles lost to each load-to-use hazard?} \hspace{1cm} 1

- $CPI_{\text{hazard}}$ due to control hazards: 
  \textit{How many cycles lost to each jump and taken branch?} \hspace{1cm} 1
Improving Processor Performance

- Increase clock frequency: deeper pipelines
- Overlap more instructions
Improving Processor Performance

- Increase clock frequency: deeper pipelines
  - Overlap more instructions

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

- Increase clock frequency: deeper pipelines
  - Overlap more instructions

- Reduce $\text{CPI}_{\text{ideal}}$: 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 Processor Performance

- Increase clock frequency: deeper pipelines
  - Overlap more instructions

- Reduce $\text{CPI}_{\text{ideal}}$: 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

- Reduce impact of control hazards: branch prediction
  - Predict both direction and target of branches and jumps
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!

- Advantage: Higher clock frequency
  - The workhorse behind multi-GHz processors
  - Intel Skylake, AMD Zen2: 19 stages, 4-5 GHz
Deeper Pipelines

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

- Advantage: Higher clock frequency
  - The workhorse behind multi-GHz processors
  - Intel Skylake, AMD Zen2: 19 stages, 4-5 GHz

- Disadvantages
  - More overlapping $\Rightarrow$ more dependencies
    - $CPI_{\text{hazard}}$ grows due to data and control hazards
  - Pipeline registers add area & power
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 $\text{CPI}_{\text{ideal}} (1/W)$
  - Skylake & Zen2: 6-wide, Power9: 8-wide
Wider (aka Superscalar) Pipelines

- Each stage operates on up to $W$ instructions each clock cycle
- Advantage: Lower $\text{CPI}_{\text{ideal}} = \frac{1}{W}$
  - Skylake & Zen2: 6-wide, Power9: 8-wide
Wider (aka Superscalar) Pipelines

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

- Advantage: Lower $\text{CPI}_{\text{ideal}} (1/W)$
  - Skylake & Zen2: 6-wide, Power9: 8-wide

- Disadvantages
  - Parallel execution $\Rightarrow$ more dependencies
    - $\text{CPI}_{\text{hazard}}$ grows due to data and control hazards
  - Much higher cost & complexity
    - More ALUs, register file ports, ...
    - Many bypass & stall cases to check

[Diagram of pipeline stages: Fetch, Decode, Read Registers, ALU, Memory, Write Registers]
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
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**

```assembly
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**
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 runs instructions as soon as their inputs become available.
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**

```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
```
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
```

**Dataflow graph**

- `ld b` - Completed
- `ld a` - Executing
- `ld c` - Not ready

```
ld b
|-|
in
ld a
|-|
in
mul 3(a-b)
|-|
in
mul ac
|-|
in
mul 7ac
|-|
in
add 3(a-b)+7ac
|-|
in
st d
```
A Modern Out-of-Order Superscalar Processor

```
I-Cache

<table>
<thead>
<tr>
<th>Branch Predict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Unit</td>
</tr>
</tbody>
</table>

Instruction Buffer

<table>
<thead>
<tr>
<th>Decode/Rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatch</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reservation Stations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int Int FP FP L/S L/S</td>
</tr>
</tbody>
</table>

Reorder Buffer

<table>
<thead>
<tr>
<th>Retire</th>
</tr>
</thead>
</table>

Write Buffer

<table>
<thead>
<tr>
<th>D-Cache</th>
</tr>
</thead>
</table>

In Order

Out Of Order

In Order

December 3, 2019

MIT 6.004 Fall 2019

L23-12
A Modern Out-of-Order Superscalar Processor

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

Reconstruct dataflow graph

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

Reconstruct dataflow graph

Execute each instruction as soon as it source operands are available

Write back results in program order

Why is this needed?
Control Hazard Penalty

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

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

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

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

Diagram:
- PC
- Fetch
- Decode
- RegRead
- Execute
- WriteBack

Next fetch started

Loose loop

Branch executed

December 3, 2019

MIT 6.004 Fall 2019

L23-13
Control Hazard 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?

Next fetch started

Loose loop

Branch executed

WriteBack

Execute

RegRead

Decode

Fetch
Control Hazard 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

Diagram:
- Fetch
- Decode
- RegRead
- Execute
- WriteBack

Next fetch started

Loose loop

Branch executed
### Control Hazard 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?

![Pipeline diagram](image-url)
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>After Inst. Decode</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>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>

December 3, 2019

MIT 6.004 Fall 2019
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>
</tbody>
</table>

Branches
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>
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 → 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
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

Predict jump/branch target and direction
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
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

Good way to improve CPI on part 3 of the design project if you use a 4-stage pipeline
Dynamic Branch Prediction

Learning from past behavior

PC -> predict -> Predictor -> Prediction

update

Truth/Feedback
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
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

BTB
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

December 3, 2019

MIT 6.004 Fall 2019
BTB Implementation Details

iMem

2^k-entry direct-mapped BTB

<table>
<thead>
<tr>
<th>tag(pc_i)</th>
<th>target_i</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

match
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
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)
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)
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)
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)
  - Not storing valid 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)
  - Not storing valid bits
- Even small BTBs are very effective!
typedef struct
    { Word pc; Word nextPc; Bool taken; } UpdateArgs;

module BTB;
    method Addr predict(Addr pc);
    input Maybe#(UpdateArgs) update default = Invalid;
endmodule
BTB Interface

typedef struct
    { Word pc; Word nextPc; Bool taken; } UpdateArgs;
module BTB;
    method Addr predict(Addr pc);
    input Maybe#(UpdateArgs) update default = Invalid;
endmodule

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

typedef struct {
    Word pc; Word nextPc; Bool taken;
} UpdateArgs;

module BTB;
    method Addr predict(Addr pc);
    input Maybe#(UpdateArgs) update default = Invalid;
endmodule

- **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.
typedef struct
    { Word pc; Word nextPc; Bool taken; } UpdateArgs;
module BTB;
    method Addr predict(Addr pc);
    input Maybe#(UpdateArgs) update default = Invalid;
endmodule

- **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 CPI on part 3 of the design project (and has lower $t_{CLK}$ than static prediction)
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:

```assembly
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</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>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?
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>
</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
Modern Processors Combine Multiple Specialized Predictors

Predict next PC immediately

Instruction type & branch/JAL target known

Branch direction & JALR target known

PC

Fetch

Decode

RegRead

Execute

WriteBack

BTB

Branch dir predictor

Return addr predictor

Loop predictor

Correct mispred
Modern Processors Combine Multiple Specialized Predictors

Predict next PC immediately

Instruction type & branch/JAL target known

Branch direction & JALR target known

Best predictors reflect program behavior
Putting It All Together: Intel Core i7 (Nehalem)

- Each core has 16 pipeline stages, ~3GHz
- 4-wide superscalar
- Out of order execution
- 2-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared

Intel, 2008, 45nm, 761M transistors, 263mm²
Putting It All Together: Intel Core i7 (Nehalem)

- Each core has 16 pipeline stages, ~3GHz
- 4-wide superscalar
- Out of order execution
- 2-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared

Intel, 2008, 45nm, 761M transistors, 263mm²
Putting It All Together: Intel Core i7 (Nehalem)

- Each core has 16 pipeline stages, ~3GHz
- 4-wide superscalar
- Out of order execution
- 2-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared
Putting It All Together: Intel Core i7 (Nehalem)

- Each core has 16 pipeline stages, ~3GHz
- 4-wide superscalar
- Out of order execution
- 2-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared
Putting It All Together: Intel Core i7 (Nehalem)

- Each core has 16 pipeline stages, ~3GHz
- 4-wide superscalar
- Out of order execution
- 2-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared
- Large overheads vs simple cores!
Putting It All Together: Intel Core i7 (Nehalem)

- Each core has 16 pipeline stages, ~3GHz
- 4-wide superscalar
- Out of order execution
- 2-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared
- Large overheads vs simple cores!
Thank you!

Good luck on Quiz 3 😊