8 Processing of control transfer instructions

- 8.1 Introduction
- 8.2 Basic approaches to branch handling
- 8.3 Delayed branching
- 8.4 Branch processing
- 8.5 Multiway branching
- 8.6 Guarded execution

8.1 Intro to Branch

- Branches modify, conditionally or unconditionally, the value of the PC.
- To transfer control
- To alter the sequence of instructions

Major types of branches

Branch: To transfer control

8.1.2 How to check the results of operations for specified conditions (branch) (e.g. equals 0, negative, and so on)

Branch: e.g.

Semantics of the non-self-explanatory instructions:

- 
- bta
- bl ta
- bcir
- bhx ta

Branch

Unconditional branches

Conditional branches

Simple unconditional branch

Branch to subroutine

Return from subroutine

Loop-closing conditional branch

Other conditional branch

Branch: e.g.

Semantics of the non-self-explanatory instructions:

- HLT Rl, ta // Branch if (Rl) ≤ 0
- SUBL Rl,R1 // Decrement R1 by 1
- BNE Rl, ta // Branch to ‘ta’ if (Rl) ≠ 0
- bl ta // Branch to ‘ta’ and store next PC into the Link Register
- bcir // Branch to the address stored in the Link Register
- bhx ta // Decrement Count Register, branch to ‘ta’ if Count Register ≠ 0

ISA concepts to implement checking for specified conditions

Result state concept

- A result state is declared to hold status information related to the results of operations (like result <0, >0 etc.).
- Typically, the result state is implemented in the form of a condition code or flags and will be updated as a part of the instruction execution.
- Conditional branch instructions can be used to interrogate this result state for specified conditions and perform a branch if the specified condition is met.

Direct check concept

- No result state is declared.
- Specified conditions are directly checked by explicit instructions.
- Conditional branching can be requested if the specified condition is met. Here, the conditional check and conditional branching can be specified either as a single instruction or as two separate instructions.
Alternatives for checking the operation results

Result state approach: Disadvantage

- The generation of the result state is not straightforward
  - It requires an irregular structure and occupies additional chip area
- The result state is a sequential concept.
  - It cannot be applied without modification in architectures which have multiple execution units.

Branch Statistics

- 20% of general-purpose code are branch
  - on average, each fifth instruction is a branch
- 5-10% of scientific code are branch
- The Majority of branches are conditional (80%)
- 75-80% of all branches are taken

Result State vs. Direct Check, e.g.

- Use multiple sets of condition codes or flags
  - It relies on programmer or compiler to use different sets condition codes or flags for different outcome generated by different EUs.
- Use Direct Check approach.

Branch statistics: Taken or Not Taken
8.1.4 The branch problem:
The delay caused in pipelining

More branch problems

- Conditional branch could cause an even longer penalty
  - evaluation of the specified condition needs an extra cycle
  - waiting for unresolved condition (the result is not yet ready)
    > e.g. wait for the result of FDIV may take 10-50 cycles
- Pipelines became more stages than 4
  - each branch would result in a yet larger number of wasted cycles (called bubbles)

8.1.5 Performance measures of branch processing

- \( P = f_t \cdot P_t + f_{nt} \cdot P_{nt} \)
  - e.g. 80386:: \( P = 0.75 \cdot 8 + 0.25 \cdot 2 = 6.5 \) cycles
  - e.g. i486:: \( P = 0.75 \cdot 2 + 0.25 \cdot 0 = 1.5 \) cycles

- Branch prediction correctly or mispredicted
  - \( P = f_c \cdot P_c + f_m \cdot P_m \)
    > e.g. Pentium:: \( P = 0.9 \cdot 0 + 0.1 \cdot 3.5 = 0.35 \) cycles

Interpretation of the concept of branch penalty

Zero-cycle branching {in no time}
8.2 Basic approaches to branch handling

Review of the basic approaches to branch handling

Speculative vs. Multiway branching

Basic scheme of delayed branching

Delayed branching: Performance Gain

- Delayed Branching: Occurrence of an unused instruction slot (unconditional branch)

- Delayed Branching: Performance Gain

• Ratio of the delay slots that can be filled with useful instructions: \( f_f \)
  \[ 60-70\% \text{ of the delay slot can be fill with useful instruction} \]
  - Fill only with: instruction that can be put in the delay slot but does not violate data dependency
  - Fill only with: instruction that can be executed in single pipeline cycle

• Frequency of branches: \( f_b \)
  \[ 20-30\% \text{ for general-propose program} \]
  \[ 5-10\% \text{ for scientific program} \]

• 100 instructions have \( 100 \times f_b \) delay slots,
  \( 100\times f_b \times f_f \) can be utilized.

• Performance Gain = \( (100\times f_b \times f_f)/100 = f_b \times f_f \)
Delayed branching: for conditional branches

- [Can be cancel or not]

Kinds of annulment for conditional branches

- No annulment (Branch with execute)
- Annul if not taken (Branch or skip)
- Annul if taken (Branch with skip)
- Annul always

Where to find the instruction to fill delay slot

- This is equivalent to the basic scheme of delayed branching
- Used e.g. for fetch-conditional branches, in order to move an instruction from within the loop body into the delay slot, as shown below
- Used e.g. for forward conditional branches, to move an instruction from the sequential path into the delay slot
- Used to provide optional delayed branching

Possible annulment options provided by architectures (use special instructions) with delayed branching (Scalar only)

<table>
<thead>
<tr>
<th>Annulment of an instruction in a delay slot</th>
<th>Branch with execute</th>
<th>Branch or skip</th>
<th>Branch with skip</th>
<th>Annul always</th>
<th>Multiplicity of delay slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 801 (1978)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>MIPS-X (1986)</td>
<td>X</td>
<td>X</td>
<td>X3</td>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>HP PA (1985)</td>
<td>X</td>
<td>X</td>
<td>X3</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>SPARC (1987)</td>
<td>X</td>
<td>X</td>
<td>X3</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>MC 88100 (1988)</td>
<td>X</td>
<td>X</td>
<td>X3</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>i380 (1983)</td>
<td>X</td>
<td>X</td>
<td>X3</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

1: Backward branches
2: Forward branches

8.4 Branch Processing: design space

- Conventional branch processing
- Layout of branch processing
- Microarchitectural implementation of branch processing

Branch detection schemes

- [early detection, better handling]

Branch detection in parallel with decoding/issuing of other instructions (in I-Buffer)
Early detection of branches by inspection of instruction that inputs to I-buffer

Integrated instruction fetch and branch detection
- Detect branch instruction during fetching
- Guess taken or not taken
- Fetch next sequential instruction or target instruction

Handling of unresolved conditional branches
- Blocking branch processing
- Speculative branch processing
- Muliway branching

-Basic kinds of branch predictions
- Fixed prediction
- True prediction
- Static prediction
- Dynamic prediction

<table>
<thead>
<tr>
<th>Processor type</th>
<th>Taken penalty cycles</th>
<th>Not-taken penalty cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC 68020 (1984)</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>MC 68030 (1987)</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>80386 (1985)</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>
The fixed prediction approach

-Always not taken vs. Always Taken

- Static branch prediction

Always not taken: Penalty figures

<table>
<thead>
<tr>
<th>Processor type</th>
<th>Taken penalty cycles</th>
<th>Not-taken penalty cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z 8000 (1984)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>S/36 (1984)</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Power (1990)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>68000 (1989)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Super68000 (1992)</td>
<td>1 (D)</td>
<td>0</td>
</tr>
<tr>
<td>Power2 (1992)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Micro68000 (1992)</td>
<td>1 (D)</td>
<td>1 (D)</td>
</tr>
</tbody>
</table>

D: Delayed branching

Penalty figures for the always taken prediction approach

<table>
<thead>
<tr>
<th>Processor type</th>
<th>Taken penalty cycles</th>
<th>Not-taken penalty cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC 68040 (1990)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Static prediction: opcode based e.g. implemented in the MC88110

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition specified</th>
<th>Bit 21 of the instr code</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x0</td>
<td>0</td>
<td>Not Taken</td>
</tr>
<tr>
<td></td>
<td>x1</td>
<td>1</td>
<td>Taken</td>
</tr>
<tr>
<td></td>
<td>a0</td>
<td>0</td>
<td>Not Taken</td>
</tr>
<tr>
<td></td>
<td>o0</td>
<td>1</td>
<td>Taken</td>
</tr>
<tr>
<td></td>
<td>o1</td>
<td>0</td>
<td>Not Taken</td>
</tr>
<tr>
<td></td>
<td>bbl (Branch on bit set)</td>
<td>0</td>
<td>Taken</td>
</tr>
<tr>
<td></td>
<td>bbl (Branch on bit clear)</td>
<td>1</td>
<td>Not Taken</td>
</tr>
</tbody>
</table>
Dynamic branch prediction: branch taken in the last n occurrences is likely to be taken next

Dynamic branch prediction: e.g.

<table>
<thead>
<tr>
<th>Device</th>
<th>1-bit</th>
<th>2-bit</th>
<th>3-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>37061</td>
<td>1990</td>
<td>1990</td>
<td>1990</td>
</tr>
<tr>
<td>MIPS</td>
<td>1995</td>
<td>1995</td>
<td>1995</td>
</tr>
</tbody>
</table>

1-bit dynamic prediction: state transition diagram

T: Branch has been taken
NT: Branch has not been taken

2-bit dynamic prediction: state transition diagram

Implicit dynamic technique

- Schemes for accessing the branch target path also used for branch prediction
- Branch Target Access Cache (BTAC)
  - holds the most recently used branch addresses
- Branch Target Instruction Cache (BTIC)
  - holds the most recently used target instructions
- BTAC or BTIC holds entries only for the taken branches
- The existence of an entry means that the corresponding branch was taken at its last occurrence
- so its next occurrence is also guessed as taken
Implementation alternatives of history bits

Combining implicit and 2-bit prediction

The effect of branch accuracy on branch penalty

Example of the implementation of the BHT

Combining implicit and 2-bit prediction

Simulation results of prediction accuracy on the SPEC
Extent of speculative processing

Recovery from a misprediction: Basic Tasks

- Discarding the result of speculative execution
- Resuming execution of the alternative path

Frequently employed schemes for shortening recovery from a misprediction

Schemes to allow or shorten recovery from a misprediction

- Basic prior measures for recovery
  - If a "taken" guess:
    - Save the sequential address
  - If a "not taken" guess:
    - Precalculate and save branch target address

Enhanced prior measures for recovery

- If a "taken" guess:
  - Save preferred sequential instructions
- If a "not taken" guess:
  - Precalculate and save branch target address, or even:
    - Prefetch branch target instructions

Extent of speculative processing: e.g.

Necessary activities to allow of to shorten recovery from a misprediction

Necessary prior measures to allow or to shorten recovery from a misprediction

- Recovery from a mispredicted taken path
- Recovery from a mispredicted sequential path

Save the address of the sequential continuation
Save preferred sequential instructions
Precalculate and save branch target address
Prefetch and save branch target instructions

shortening recovery from a misprediction: needs

Use of two addresses per speculated conditional branch
Use of two or three instruction buffers in superscalar processors

K6-266 (1995)
Using two instruction buffers in the supersparc to shorten recovery from a misprediction: e.g.

Using three instruction buffers in the Nx586 to shorten recovery from a misprediction: e.g.

8.4.5 Branch penalty for taken guesses depends on branch target accessing schemes

-Compute/fetch scheme for accessing branch targets (IFAR vs. PC)

-BTAC scheme for accessing branch targets (associative search for BA, if found get BTA) [0-cycle branch: BA=BA-4]

-BTIC scheme: store next BTA
- BTIC scheme: calculate next BTA

- Successor index in the I-cache scheme to access the branch target path (index: next I, or target I)

Successor index in the I-cache scheme: e.g.
The microarchitecture of the UltraSparc

Predecode unit: detects branches, BTA, make predictions (based on compiler’s hint bit), set up I-cache Next address

Branch target accessing trends

8.5 Multiway branching: {two IFA’s or PC’s}
Threefold multiway branching: only one correct path!

8.6 Guarded Execution

- a means to eliminate branches
- by conditional operate instructions
  - IF the condition associated with the instruction is met,
  - THEN perform the specified operation
  - ELSE do not perform the operation
- e.g. original
  - beg r1, label  // if (r1) = 0 branch to label
  - move r2, r3  // move (r2) into r3
  - label: …
- e.g. guarded
  - cmovne r1, r2, r3  // if (r1) != 0, move (r2) into r3
  - …
- Convert control dependencies into data dependencies

Eliminated branches by full and restricted guarding
(full: all instruction guarded, restricted: ALU inst guarded)

<table>
<thead>
<tr>
<th>Program</th>
<th>Percentage of jump branches (%)</th>
<th>Percentage of eliminated branches (%)</th>
<th>Full guarding</th>
<th>Restricted guarding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>26.44</td>
<td>24.88</td>
<td>84.29</td>
<td>18.24</td>
</tr>
<tr>
<td>Egmont</td>
<td>29.07</td>
<td>44.55</td>
<td>54.98</td>
<td>40.04</td>
</tr>
<tr>
<td>Espresso</td>
<td>38.68</td>
<td>16.78</td>
<td>29.03</td>
<td>10.17</td>
</tr>
<tr>
<td>Gimp2</td>
<td>24.34</td>
<td>51.92</td>
<td>17.04</td>
<td>9.64</td>
</tr>
<tr>
<td>Ge</td>
<td>24.63</td>
<td>43.07</td>
<td>17.74</td>
<td>9.83</td>
</tr>
<tr>
<td>Snobench</td>
<td>15.70</td>
<td>35.65</td>
<td>47.10</td>
<td>11.35</td>
</tr>
<tr>
<td>Supercas</td>
<td>5.65</td>
<td>50.69</td>
<td>19.16</td>
<td>11.15</td>
</tr>
<tr>
<td>Tightpix</td>
<td>16.83</td>
<td>37.53</td>
<td>41.60</td>
<td>17.08</td>
</tr>
<tr>
<td>TIUX</td>
<td>25.69</td>
<td>12.80</td>
<td>24.03</td>
<td>5.99</td>
</tr>
<tr>
<td>Thimturk</td>
<td>11.57</td>
<td>62.31</td>
<td>53.79</td>
<td>25.36</td>
</tr>
<tr>
<td>Tycho</td>
<td>18.28</td>
<td>13.64</td>
<td>33.84</td>
<td>7.10</td>
</tr>
<tr>
<td>Xlop</td>
<td>27.03</td>
<td>13.94</td>
<td>14.33</td>
<td>13.87</td>
</tr>
<tr>
<td>Yacc</td>
<td>20.64</td>
<td>19.53</td>
<td>38.95</td>
<td>8.18</td>
</tr>
<tr>
<td>Arithmetic Mean</td>
<td>23.37</td>
<td>33.15</td>
<td>35.87</td>
<td>14.76</td>
</tr>
</tbody>
</table>

Guarded Execution: Disadvantages

- guarding transforms instructions from both the taken and the not-taken paths into guard instruction
  - increase number of instructions
  - by 33% for full guarding
  - by 8% for restricted guarding
  - [more instructions more time and space]
- guarding requires additional hardware resources if an increase in processing time is to be avoided
  - VLIW