Instruction Level Parallelism
Predication & Precise Exceptions

Review: Importance of The Branch Problem

- Assume a 5-wide *superscalar* pipeline with 20-cycle branch resolution latency

- How long does it take to fetch 500 instructions?
  - Assume no fetch breaks and 1 out of 5 instructions is a branch
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work
  - 99% accuracy
    - 100 (correct path) + 20 (wrong path) = 120 cycles
    - 20% extra instructions fetched
  - 98% accuracy
    - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
    - 40% extra instructions fetched
  - 95% accuracy
    - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
    - 100% extra instructions fetched
Review: Local and Global Branch Prediction

- Last-time and 2BC predictors exploit "last-time" predictability

- Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
  - Global branch correlation

- Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
  - Local branch correlation

Review: Hybrid Branch Prediction in Alpha 21264

- Minimum branch penalty: 7 cycles
- Typical branch penalty: 11+ cycles
- 48K bits of target addresses stored in I-cache
- Predictor tables are reset on a context switch
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:
  - **Stall** the pipeline until we know the next fetch address
  - Guess the next fetch address (**branch prediction**)
  - Employ delayed branching (**branch delay slot**)
  - Do something else (**fine-grained multithreading**)
  - Eliminate control-flow instructions (**predicated execution**)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (**multipath execution**)

Review: Predicate Combining (**not** Predicated Execution)

- Complex predicates are converted into multiple branches
  - if ((a == b) && (c < d) && (a > 5000)) { ... }  
    - 3 conditional branches
- Problem: This increases the number of control dependencies
- Idea: Combine predicate operations to feed a single branch instruction
  - Predicates stored and operated on using condition registers
  - A single branch checks the value of the combined predicate
- Fewer branches in code $\rightarrow$ fewer mispredictions/stalls
- Possibly unnecessary work
  - If the first predicate is false, no need to compute other predicates
- Condition registers exist in IBM RS6000 and the POWER architecture
**Predication (Predicated Execution)**

- **Idea:** Compiler converts control dependence into data dependence → branch is eliminated
  - Each instruction has a predicate bit set based on the predicate computation
  - Only instructions with TRUE predicates are committed (others turned into NOPs)

(normal branch code)  (predicated code)

```
if (cond) {
  b = 0;
} else {
  b = 1;
}
```

```
if (cond) {
  b = 0;
} else {
  b = 1;
}
```

**Conditional Move Operations**

- Very limited form of predicated execution

- **CMOV R1 ← R2**
  - \( R1 = (\text{ConditionCode} == \text{true}) ? \text{R2} : \text{R1} \)
  - Employed in most modern ISAs (x86, Alpha)
Review: CMOV Operation

- Suppose we had a Conditional Move instruction...
  - CMOV condition, \( R1 \leftarrow R2 \)
  - \( R1 = (\text{condition} == \text{true}) \ ? \ R2 : R1 \)
  - Employed in most modern ISAs (x86, Alpha)

- Code example with branches vs. CMOVs

\[
\begin{align*}
\text{if (} a == 5 \text{)} \{ b = 4; \} & \text{ else } \{ b = 3; \} \\
\text{CMPEQ condition, } a, 5; & \\
\text{CMOV condition, } b \leftarrow 4; & \\
\text{CMOV !condition, } b \leftarrow 3;
\end{align*}
\]

Predicated Execution (II)

- Predicated execution can be high performance and energy-efficient

```
Predicated Execution
Fetch Decode Rename Schedule RegisterRead Execute

Branch Prediction
Fetch Decode Rename Schedule RegisterRead Execute
```

Pipeline flush!!
Predicated Execution (III)

**Advantages:**
+ Eliminates mispredictions for hard-to-predict branches
+ No need for branch prediction for some branches
+ Good if misprediction cost > useless work due to predication
+ Enables code optimizations hindered by the control dependency
+ Can move instructions more freely within predicated code

**Disadvantages:**
-- Causes useless work for branches that are easy to predict
-- Reduces performance if misprediction cost < useless work
-- Adaptivity: Static predication is not adaptive to run-time branch behavior. Branch behavior changes based on input set, phase, control-flow path.
-- Additional hardware and ISA support
-- Cannot eliminate all hard to predict branches
-- Loop branches?

---

Predicated Execution in Intel Itanium

- Each instruction can be separately predicated
- 64 one-bit predicate registers
  
  each instruction carries a 6-bit predicate field
- An instruction is effectively a NOP if its predicate is false

![Diagram showing predicated execution in Intel Itanium](image-url)
Conditional Execution in ARM ISA

- Almost all ARM instructions can include an optional condition code.

- An instruction with a condition code is only executed if the condition code flags in the CPSR meet the specified condition.
Conditional Execution in ARM ISA

<table>
<thead>
<tr>
<th>Cond</th>
<th>31</th>
<th>28</th>
<th>24</th>
<th>20</th>
<th>16</th>
<th>12</th>
<th>8</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>=</td>
<td>EQ</td>
<td>Z</td>
<td>set</td>
<td>(equal)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>=</td>
<td>NE</td>
<td>Z</td>
<td>clear</td>
<td>(not equal)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>=</td>
<td>HS</td>
<td>CS</td>
<td>C</td>
<td>set</td>
<td>(unsigned higher or same)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>=</td>
<td>LO</td>
<td>CC</td>
<td>C</td>
<td>clear</td>
<td>(unsigned lower)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>=</td>
<td>MI</td>
<td>N</td>
<td>set</td>
<td>(negative)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>=</td>
<td>PL</td>
<td>N</td>
<td>clear</td>
<td>(positive or zero)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>=</td>
<td>VS</td>
<td>V</td>
<td>set</td>
<td>(overflow)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>=</td>
<td>VC</td>
<td>V</td>
<td>clear</td>
<td>(no overflow)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>=</td>
<td>HI</td>
<td>C</td>
<td>set</td>
<td>and</td>
<td>Z</td>
<td>clear</td>
<td>(unsigned higher)</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>=</td>
<td>LS</td>
<td>C</td>
<td>clear</td>
<td>or</td>
<td>Z</td>
<td>(set unsigned lower or same)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>=</td>
<td>GE</td>
<td>N</td>
<td>set</td>
<td>and</td>
<td>V</td>
<td>set</td>
<td>(&gt; or =)</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>=</td>
<td>LT</td>
<td>N</td>
<td>set</td>
<td>and</td>
<td>V</td>
<td>clear</td>
<td>(&gt;)</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>=</td>
<td>GT</td>
<td>Z</td>
<td>clear</td>
<td>and</td>
<td>either</td>
<td>N</td>
<td>set</td>
<td>and</td>
</tr>
<tr>
<td>1101</td>
<td>=</td>
<td>LE</td>
<td>Z</td>
<td>set</td>
<td>or</td>
<td>N</td>
<td>set</td>
<td>and</td>
<td>V</td>
</tr>
<tr>
<td>1110</td>
<td>=</td>
<td>AL</td>
<td>-</td>
<td>always</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>=</td>
<td>NV</td>
<td>-</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Conditional Execution in ARM ISA

* To execute an instruction conditionally, simply postfix it with the appropriate condition:
  - For example an add instruction takes the form:
    - `ADD r0, r1, r2 ; r0 = r1 + r2` (ADDAL)
  - To execute this only if the zero flag is set:
    - `ADDEQ r0, r1, r2 ; If zero flag set then...
      ... r0 = r1 + r2`

* By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.
  - For example to add two numbers and set the condition flags:
    - `ADDS r0, r1, r2 ; r0 = r1 + r2
      ... and set flags`
Conditional Execution in ARM ISA

* Convert the GCD algorithm given in this flowchart into
  1) “Normal” assembler, where only branches can be conditional.
  2) ARM assembler, where all instructions are conditional, thus improving code density.

* The only instructions you need are CMP, B and SUB.

---

Conditional Execution in ARM ISA

“Normal” Assembler

god cmp r0, r1 ; reached the end?
    beq stop
    blt less ; if r0 > r1
    sub r0, r0, r1 ; subtract r1 from r0
    bal gcd
less sub r1, r1, r0 ; subtract r0 from r1
    bal gcd
stop

ARM Conditional Assembler

god cmp r0, r1 ; if r0 > r1
    subgt r0, r0, r1 ; subtract r1 from r0
    sublt r1, r1, r0 ; else subtract r0 from r1
    bne gcd ; reached the end?
Idealism

- Wouldn’t it be nice
  - If the branch is eliminated (predicated) when it will actually be mispredicted
  - If the branch were predicted when it will actually be correctly predicted

- Wouldn’t it be nice
  - If predication did not require ISA support

Improving Predicated Execution

- Three major limitations of predication
  1. *Adaptivity*: non-adaptive to branch behavior
  2. *Complex CFG*: inapplicable to loops/complex control flow graphs
  3. *ISA*: Requires large ISA changes

- Dynamic Predicated Execution
Wish Branches

- The compiler generates code (with wish branches) that can be executed either as predicated code or non-predicated code (normal branch code).
- The hardware decides to execute predicated code or normal branch code at run-time based on the confidence of branch prediction.
- Easy to predict: normal branch code
- Hard to predict: predicated code
Wish Branches vs. Predicated Execution

- Advantages compared to predicated execution
  - Reduces the overhead of predication
  - Increases the benefits of predicated code by allowing the compiler to generate more aggressively-predicated code
  - Makes predicated code less dependent on machine configuration (e.g. branch predictor)

- Disadvantages compared to predicated execution
  - Extra branch instructions use machine resources
  - Extra branch instructions increase the contention for branch predictor table entries
  - Constrains the compiler’s scope for code optimizations

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (branch prediction)
  - Employ delayed branching (branch delay slot)
  - Do something else (fine-grained multithreading)
  - Eliminate control-flow instructions (predicated execution)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)
Multi-Path Execution

- **Idea:** Execute both paths after a conditional branch
  - For all branches
  - For a hard-to-predict branch: Use dynamic confidence estimation

- **Advantages:**
  + Improves performance if misprediction cost > useless work
  + No ISA change needed

- **Disadvantages:**
  -- What happens when the machine encounters another hard-to-predict branch? Execute both paths again?
  -- Paths followed quickly become exponential
  -- Each followed path requires its own registers, PC, GHR
  -- Wasted work (and reduced performance) if paths merge

---

Dual-Path Execution versus Predication
Remember: Branch Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Direction at fetch time</th>
<th>Number of possible next fetch addresses?</th>
<th>When is next fetch address resolved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>Unknown</td>
<td>2</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Unconditional</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Call</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Return</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Indirect</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
</tbody>
</table>

Different branch types can be handled differently

Call and Return Prediction

- **Direct calls are easy to predict**
  - Always taken, single target
  - Call marked in BTB, target predicted by BTB

- **Returns are indirect branches**
  - A function can be called from many points in code
  - A return instruction can have many target addresses
    - Next instruction after each call point for the same function
  - Observation: Usually a return matches a call
  - Idea: Use a stack to predict return addresses (Return Address Stack)
    - A fetched call: pushes the return (next instruction) address on the stack
    - A fetched return: pops the stack and uses the address as its predicted target
    - Accurate most of the time: 8-entry stack $\rightarrow > 95\%$ accuracy
Indirect Branch Prediction (I)

- Register-indirect branches have multiple targets

- Used to implement
  - Switch-case statements
  - Virtual function calls
  - Jump tables (of function pointers)
  - Interface calls

Indirect Branch Prediction (II)

- Idea 1: Predict the last resolved target as the next fetch address
  - Simple: Use the BTB to store the target address
  - Inaccurate: 50% accuracy (empirical). Many indirect branches switch between different targets

- Idea 2: Use history based target prediction
  - E.g., Index the BTB with GHR XORed with Indirect Branch PC
  - More accurate
  - An indirect branch maps to (too) many entries in BTB
    - Conflict misses with other branches (direct or indirect)
    - Inefficient use of space if branch has few target addresses