18-447
Computer Architecture
Lecture 9: Branch Handling and Branch Prediction

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 2/3/2014
Readings for Next Few Lectures (I)

- P&H Chapter 4.9-4.11

  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts


Readings for Next Few Lectures (II)

Data Dependence Handling: More Depth & Implementation
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**)
Delayed Branching (I)

- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles

- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are always executed regardless of branch direction.

- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions

- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot
Delayed Branching (II)

Normal code:

Timeline:

Delayed branch code:

Timeline:

6 cycles

5 cycles
Fancy Delayed Branching (III)

- Delayed branch with squashing
  - In SPARC
  - If the branch falls through (not taken), the delay slot instruction is not executed
  - Why could this help?

Normal code:          Delayed branch code:          Delayed branch w/ squashing:

```
X: A
  B
  C
  BC X
  D
  E
```

```
X: A
  B
  C
  BC X
  NOP
  D
  E
```

```
X: A
  B
  C
  BC X
  A
  D
  E
```
Delayed Branching (IV)

Advantages:
+ Keeps the pipeline full with useful instructions in a simple way assuming
  1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves
  2. All delay slots can be filled with useful instructions

Disadvantages:
-- Not easy to fill the delay slots (even with a 2-stage pipeline)
  1. Number of delay slots increases with pipeline depth, superscalar execution width
  2. Number of delay slots should be variable with variable latency operations. Why?
-- Ties ISA semantics to hardware implementation
  -- SPARC, MIPS, HP-PA: 1 delay slot
  -- What if pipeline implementation changes with the next design?
An Aside: Filling the Delay Slot

reordering data independent (RAW, WAW, WAR) instructions does not change program semantics

within same basic block

if $s2 = 0$ then

$add \ s1, \ s2, \ s3$

_{Delay slot_}

Becomes

if $s1 = 0$ then

$add \ s1, \ s2, \ s3$

_{Delay slot_}

Becomes

if $s1 = 0$ then

$sub \ t4, \ t5, \ t6$

_{Delay slot_}

Becomes

if $s2 = 0$ then

$add \ s1, \ s2, \ s3$

For correctness: a new instruction added to not-taken path??

For correctness: a new instruction added to taken path??

For correctness: a new instruction added to not-taken path??

Safe?
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*)
Fine-Grained Multithreading

- **Idea:** Hardware has multiple thread contexts. Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread.
  - Branch/instruction resolution latency overlapped with execution of other threads’ instructions.

- No logic needed for handling control and data dependences within a thread.
  - Single thread performance suffers.
  - Extra logic for keeping thread contexts.
  - Does not overlap latency if not enough threads to cover the whole pipeline.
Fine-grained Multithreading

- **Idea:** Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently.

- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads.

- Improves pipeline utilization by taking advantage of multiple threads.


Fine-grained Multithreading: History

- CDC 6600’s peripheral processing unit is fine-grained multithreaded
  - Processor executes a different I/O thread every cycle
  - An operation from the same thread is executed every 10 cycles

- Denelcor HEP (Heterogeneous Element Processor)
  - 120 threads/processor
  - available queue vs. unavailable (waiting) queue for threads
  - each thread can only have 1 instruction in the processor pipeline; each thread independent
  - to each thread, processor looks like a non-pipelined machine
  - system throughput vs. single thread performance tradeoff
Fine-grained Multithreading in HEP

- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
  - assuming no memory access
Multithreaded Pipeline Example

Slide credit: Joel Emer
Sun Niagara Multithreaded Pipeline

Fine-grained Multithreading

**Advantages**

- No need for dependency checking between instructions (only one instruction in pipeline from a single thread)
- No need for branch prediction logic
- Otherwise-bubble cycles used for executing useful instructions from different threads
- Improved system throughput, latency tolerance, utilization

**Disadvantages**

- Extra hardware complexity: multiple hardware contexts, thread selection logic
- Reduced single thread performance (one instruction fetched every N cycles)
- Resource contention between threads in caches and memory
- Some dependency checking logic between threads remains (load/store)
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)
Branch Prediction: Guess the Next Instruction to Fetch
Misprediction Penalty

LD R1, MEM[R0]
ADD R2, R2, #1
BRzero 0x0001
ADD R3, R2, #1
MUL R1, R2, R3
LD R2, MEM[R2]
LD R0, MEM[R2]

PC 0x0001
I-$ 0x0007$
DEC 0x0006
RF 0x0005
D-$ 0x0004$
WB 0x0003

Misprediction penalty occurs when a branch mispredicts, leading to a penalty in the pipeline stages.
Branch Prediction

- Processors are pipelined to increase concurrency
- How do we keep the pipeline full in the presence of branches?
  - Guess the next instruction when a branch is fetched
  - Requires guessing the direction and target of a branch

What to fetch next?

Pipeline

1. Fetch
2. Decode
3. Rename
4. Schedule
5. Register Read
6. Execute

Target Misprediction Detected! Flush the pipeline

Branch condition, TARGET
Branch Prediction: Always PC+4

When a branch resolves:
- branch target (Inst$_k$) is fetched
- all instructions fetched since inst$_h$ (so called “wrong-path” instructions) must be flushed.
Pipeline Flush on a Misprediction

Inst_h is a branch
Performance Analysis

- correct guess ⇒ no penalty  ~86% of the time
- incorrect guess ⇒ 2 bubbles

Assume
- no data hazards
- 20% control flow instructions
- 70% of control flow instructions are taken
- CPI = \[ 1 + (0.20\times0.7) \times 2 \] =
  \[ = [ 1 + 0.14 \times 2 ] = 1.28 \]

Can we reduce either of the two penalty terms?

probability of penalty for
a wrong guess a wrong guess
Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early

\[
\text{CPI} = [1 + (0.2 \times 0.7) \times 1] = 1.14
\]
Branch Prediction (Enhanced)

- **Idea:** Predict the next fetch address (to be used in the next cycle)

- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)

- **Observation:** Target address remains the same for a conditional direct branch across dynamic instances
  - **Idea:** Store the target address from previous instance and access it with the PC
  - Called **Branch Target Buffer (BTB)** or Branch Target Address Cache
Fetch Stage with BTB and Direction Prediction

Direction predictor (2-bit counters)

Cache of Target Addresses (BTB: Branch Target Buffer)

Always taken CPI = \[ 1 + (0.20 \times 0.3) \times 2 \] = 1.12 (70% of branches taken)
More Sophisticated Branch Direction Prediction

Direction predictor (2-bit counters)

Which direction earlier branches went

Global branch history

Program Counter

Address of the current branch

XOR

PC + inst size

hit?

taken?

Next Fetch Address

Cache of Target Addresses (BTB: Branch Target Buffer)
Simple Branch Direction Prediction Schemes

- **Compile time (static)**
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)

- **Run time (dynamic)**
  - Last time prediction (single-bit)
More Sophisticated Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid
Static Branch Prediction (I)

- **Always not-taken**
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40%
  - Compiler can layout code such that the likely path is the “not-taken” path

- **Always taken**
  - No direction prediction
  - Better accuracy: ~60-70%
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC

- **Backward taken, forward not taken (BTFN)**
  - Predict backward (loop) branches as taken, others not-taken
Static Branch Prediction (II)

- Profile-based
  - Idea: Compiler determines likely direction for each branch using profile run. Encodes that direction as a hint bit in the branch instruction format.

+ Per branch prediction (more accurate than schemes in previous slide) $\rightarrow$ accurate if profile is representative!

- Requires hint bits in the branch instruction format

- Accuracy depends on dynamic branch behavior:
  - TTTTTTTTTTNNNNNNNNNN $\rightarrow$ 50% accuracy
  - TNTNTNTNTNTNTNTNTNTNTNTNTNTNTN $\rightarrow$ 50% accuracy

- Accuracy depends on the representativeness of profile input set
Program-based (or, program analysis based)

- Idea: Use heuristics based on program analysis to determine statically-predicted direction
- Opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
- Loop heuristic: Predict a branch guarding a loop execution as taken (i.e., execute the loop)
- Pointer and FP comparisons: Predict not equal

+ Does not require profiling
-- Heuristics might be not representative or good
-- Requires compiler analysis and ISA support


- 20% misprediction rate
Static Branch Prediction (III)

- Programmer-based
  - Idea: Programmer provides the statically-predicted direction
  - Via pragmas in the programming language that qualify a branch as likely-taken versus likely-not-taken

+ Does not require profiling or program analysis
+ Programmer may know some branches and their program better than other analysis techniques

-- Requires programming language, compiler, ISA support
-- Burdens the programmer?
Aside: Pragmas

- **Idea**: Keywords that enable a programmer to convey hints to lower levels of the transformation hierarchy

- if (likely(x)) { ... }
- if (unlikely(error)) { ... }

- Many other hints and optimizations can be enabled with pragmas
  - E.g., whether a loop can be parallelized
  - `#pragma omp parallel`
  - **Description**
    - The `omp parallel` directive explicitly instructs the compiler to parallelize the chosen segment of code.
Static Branch Prediction

- All previous techniques can be combined
  - Profile based
  - Program based
  - Programmer based

- How would you do that?

- What are common disadvantages of all three techniques?
  - Cannot adapt to dynamic changes in branch behavior
    - This can be mitigated by a dynamic compiler, but not at a fine granularity (and a dynamic compiler has its overheads...)
Dynamic Branch Prediction

- Idea: Predict branches based on dynamic information (collected at run-time)

- Advantages
  + Prediction based on history of the execution of branches
  + It can adapt to dynamic changes in branch behavior
  + No need for static profiling: input set representativeness problem goes away

- Disadvantages
  -- More complex (requires additional hardware)
Last Time Predictor

- **Last time predictor**
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
    
    
    TTTTTTTTTTNNNNNNNNNNN \rightarrow 90\% accuracy

- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

+ Loop branches for loops with large number of iterations

-- Loop branches for loops with small number of iterations

    TNTNTNTNTNTNTNTNTNTNTNTNTN \rightarrow 0\% accuracy

Last-time predictor CPI = [ 1 + (0.20*0.15) * 2 ] = 1.06  (Assuming 85\% accuracy)
Implementing the Last-Time Predictor

The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch.
State Machine for Last-Time Prediction

![State Machine Diagram]
Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from \(T \rightarrow NT\) or \(NT \rightarrow T\) too quickly
  - even though the branch may be mostly taken or mostly not taken

- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for \(T\) or \(NT\) instead of 1 state for each

Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

Accuracy for a loop with N iterations = \((N-1)/N\)

\[ \text{TNTNTNTNTNTNTNTNTNTNTN} \rightarrow \text{50\% accuracy} \]

(assuming init to weakly taken)

+ Better prediction accuracy
  \[ 2\text{BC predictor CPI} = [1 + \left(0.20 \times 0.10\right) \times 2] = 1.04 \] (90\% accuracy)

-- More hardware cost (but counter can be part of a BTB entry)
State Machine for 2-bit Saturating Counter

- Counter using saturating arithmetic
  - There is a symbol for maximum and minimum values
Hysteresis Using a 2-bit Counter

Change prediction after 2 consecutive mistakes
Is This Enough?

- ~85-90% accuracy for many programs with 2-bit counter based prediction (also called bimodal prediction)

- Is this good enough?

- How big is the branch problem?
Rethinking the The Branch Problem

- Control flow instructions (branches) are frequent
  - 15-25% of all instructions

- Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
  - N cycles: (minimum) branch resolution latency
  - Stalling on a branch wastes instruction processing bandwidth (i.e. reduces IPC)
    - N x IW instruction slots are wasted (IW: issue width)

- How do we keep the pipeline full after a branch?
- Problem: Need to determine the next fetch address when the branch is fetched (to avoid a pipeline bubble)
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
Can We Do Better?

- 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
Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path is correlated with the outcome of the next branch

\[
\begin{align*}
\text{if (cond1)} \\
\ldots \\
\text{if (cond1 AND cond2)}
\end{align*}
\]

- If first branch not taken, second also not taken

```
branch Y: if (cond1) a = 2; \\
\ldots \\
branch X: if (a == 0)
```

- If first branch taken, second definitely not taken
Global Branch Correlation (II)

branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if (cond1 AND cond2)

- If Y and Z both taken, then X also taken
- If Y or Z not taken, then X also not taken
Global Branch Correlation (III)

- Eqntott, SPEC 1992

```java
if (aa==2) ;; B1
    aa=0;
if (bb==2) ;; B2
    bb=0;
if (aa!=bb) {
    ....
}
```

If **B1** is not taken (i.e. aa==0@B3) and **B2** is not taken (i.e. bb=0@B3) then **B3** is certainly taken
Capturing Global Branch Correlation

- Idea: Associate branch outcomes with “global T/NT history” of all branches
- Make a prediction based on the outcome of the branch the last time the same global branch history was encountered

Implementation:
- Keep track of the “global T/NT history” of all branches in a register → Global History Register (GHR)
- Use GHR to index into a table of that recorded the outcome that was seen for that GHR value in the recent past → Pattern History Table (table of 2-bit counters)

- Global history/branch predictor
- Uses two levels of history (GHR + history at that GHR)
Two Level Global Branch Prediction

- First level: **Global branch history register** (N bits)
  - The direction of last N branches
- Second level: **Table of saturating counters for each history entry**
  - The direction the branch took the last time the same history was seen

![Pattern History Table (PHT)](image)

How Does the Global Predictor Work?

```plaintext
for (i=0; i<100; i++)
    for (j=0; j<3; j++)
```

After the initial startup time, the conditional branches have the following behavior, assuming GR is shifted to the left:

<table>
<thead>
<tr>
<th>test</th>
<th>value</th>
<th>GR</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>j&lt;3</td>
<td>j=1</td>
<td>1101</td>
<td>taken</td>
</tr>
<tr>
<td>j&lt;3</td>
<td>j=2</td>
<td>1011</td>
<td>taken</td>
</tr>
<tr>
<td>j&lt;3</td>
<td>j=3</td>
<td>0111</td>
<td>not taken</td>
</tr>
<tr>
<td>i&lt;100</td>
<td>i=3</td>
<td>1110</td>
<td>usually taken</td>
</tr>
</tbody>
</table>

Intel Pentium Pro Branch Predictor

- 4-bit global history register
- Multiple pattern history tables (of 2 bit counters)
  - Which pattern history table to use is determined by lower order bits of the branch address
Improving Global Predictor Accuracy

- Idea: Add more context information to the global predictor to take into account which branch is being predicted
  - Gshare predictor: GHR hashed with the Branch PC
    - More context information
    - Better utilization of PHT
    - Increases access latency

One-Level Branch Predictor

Direction predictor (2-bit counters)

Program Counter

Address of the current instruction

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address

PC + inst size

hit?

taken?
Two-Level Global History Predictor

Direction predictor (2-bit counters)

Which direction earlier branches went

Global branch history

Program Counter

Address of the current instruction

Cache of Target Addresses (BTB: Branch Target Buffer)

PC + inst size

Next Fetch Address

taken?

hit?

target address
Two-Level Gshare Predictor

Which direction earlier branches went

Global branch history

Program Counter

Address of the current instruction

Direction predictor (2-bit counters)

taken?

PC + inst size

hit?

target address

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address
Can We Do Better?

- 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
Local Branch Correlation

for (i=1; i<=4; i++) { }

If the loop test is done at the end of the body, the corresponding branch will execute the pattern \((1110)^n\), where 1 and 0 represent taken and not taken respectively, and \(n\) is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

Capturing Local Branch Correlation

- **Idea:** Have a per-branch history register
  - Associate the predicted outcome of a branch with “T/NT history” of the same branch
- Make a prediction is based on the outcome of the branch the last time the same local branch history was encountered

- Called the local history/branch predictor
- Uses two levels of history (Per-branch history register + history at that history register value)
Two Level Local Branch Prediction

- First level: A set of local history registers (N bits each)
  - Select the history register based on the PC of the branch
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen

Two-Level Local History Predictor

Which directions earlier instances of *this branch* went

Direction predictor (2-bit counters)

Program Counter

Address of the current instruction

Cache of Target Addresses (BTB: Branch Target Buffer)

PC + inst size

hit?

taken?

Next Fetch Address

target address
Hybrid Branch Predictors

- **Idea:** Use more than one type of predictor (i.e., multiple algorithms) and select the “best” prediction
  - E.g., hybrid of 2-bit counters and global predictor

- **Advantages:**
  - Better accuracy: different predictors are better for different branches
  - Reduced *warmup* time (faster-warmup predictor used until the slower-warmup predictor warms up)

- **Disadvantages:**
  - Need “meta-predictor” or “selector”
  - Longer access latency

Alpha 21264 Tournament Predictor

- 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

Branch Prediction Accuracy (Example)

- Bimodal: table of 2bc indexed by branch address

Figure 13: Combined Predictor Performance by Benchmark
Biased Branches

- **Observation:** Many branches are biased in one direction (e.g., 99% taken)

- **Problem:** These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing “interference” in branch prediction tables and history registers

- **Solution:** Detect such biased branches, and predict them with a simpler predictor

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 → fewer mipredictions/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)

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

(predicated code)

```
if (cond) {
  p1 = (cond)
  branch p1, TARGET
}
else {
  p1 = (cond)
  (!p1) mov b, 1
  jmp JOIN
}
TARGET:
  mov b, 0
add   x, b, 1
```
Conditional Move Operations

- Very limited form of predicated execution

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

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

- Code example with branches vs. CMOVs
  if (a == 5) {b = 4;} else {b = 3;}

  CMPEQ condition, a, 5;
  CMOV condition, b ← 4;
  CMOV !condition, b ← 3;
Predicated Execution (II)

- Predicated execution can be high performance and energy-efficient

![Diagram showing predicated execution and branch prediction with pipeline flush]

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
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>001</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
<td>0000000 AS</td>
<td>RD</td>
<td>RN</td>
<td>Rs</td>
<td>1001</td>
<td>Rm</td>
</tr>
<tr>
<td>Cond</td>
<td>000011U AS</td>
<td>RdHi</td>
<td>RdLo</td>
<td>Rs</td>
<td>1001</td>
<td>Rm</td>
</tr>
<tr>
<td>Cond</td>
<td>00010B00</td>
<td>RN</td>
<td>RD</td>
<td>00001001</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>011IPUBLWL</td>
<td>RN</td>
<td>RD</td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>100PUSWL</td>
<td>RN</td>
<td>Register List</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>000PUIWL</td>
<td>RN</td>
<td>Rd</td>
<td>Offset1 1SH1 Offset2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>000PU0WL</td>
<td>RN</td>
<td>Rd</td>
<td>000010SH1 Rm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>101L</td>
<td>Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>0001001011111111111111110001</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>110PUNWL</td>
<td>RN</td>
<td>CRd</td>
<td>CPNum</td>
<td>Offset</td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>1110 CRn</td>
<td>CRd</td>
<td>CPNum</td>
<td>Op2</td>
<td>0</td>
<td>CRm</td>
</tr>
<tr>
<td>Cond</td>
<td>1110 CRnL</td>
<td>CRn</td>
<td>Rd</td>
<td>CPNum</td>
<td>Op2</td>
<td>1</td>
</tr>
<tr>
<td>Cond</td>
<td>1111</td>
<td>SWI Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction type
- Data processing / PSR Transfer
- Multiply
- Long Multiply (v3M/v4 only)
- Swap
- Load/Store Byte/Word
- Load/Store Multiple
- Halfword transfer: Immediate offset (v4 only)
- Halfword transfer: Register offset (v4 only)
- Branch
- Branch Exchange (v4T only)
- Coprocessor data transfer
- Coprocessor data operation
- Coprocessor register transfer
- Software interrupt
Conditional Execution in ARM ISA

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

The ARM Instruction Set - ARM University Program - V1.0
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 (ADDA)
  
  - 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

```
gcd    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

```
gcd    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

- **Wish Branches** [Kim+, MICRO 2005]
  - Solve 1 and partially 2 (for loops)

- **Dynamic Predicated Execution**
  - Diverge-Merge Processor [Kim+, MICRO 2006]
    - Solves 1, 2 (partially), 3
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 Jump/Join

normal branch code

predicated code

A

T

N

C

D

p1 = (cond)
branch p1, TARGET

mov b, 1
jmp JOIN

TARGET:
mov b,0

A

B

C

D

p1 = (cond)

(!p1) mov b,1

wished join!
p1
JOIN:

nop

wished jump

A

B

C

D

p1 = (cond)

(!p1) mov b,1

wished jump!

NOP

A

B

C

D

TARGET:

wished join!

nop

High Confidence

Low Confidence
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 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

Hard to predict

Dual-path

Predicated Execution

path 1

path 2
### 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

```
TARG  A
    T   N
   A+1
```

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

```
A
br.cond TARGET

R1 = MEM[R2]
branch R1
```

Conditional (Direct) Branch

Indirect Jump
Indirect Branch Prediction (II)

- No direction prediction needed
- 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
Issues in Branch Prediction (I)

- Need to identify a branch before it is fetched

- How do we do this?
  - BTB hit → indicates that the fetched instruction is a branch
  - BTB entry contains the “type” of the branch

- What if no BTB?
  - Bubble in the pipeline until target address is computed
  - E.g., IBM POWER4
Issues in Branch Prediction (II)

- **Latency**: Prediction is latency critical
  - Need to generate next fetch address for the next cycle
  - Bigger, more complex predictors are more accurate but slower
Complications in Superscalar Processors

- “Superscalar” processors
  - attempt to execute more than 1 instruction-per-cycle
  - must fetch multiple instructions per cycle

Consider a 2-way superscalar fetch scenario

- (case 1) Both insts are not taken control flow inst
  - \[ nPC = PC + 8 \]

- (case 2) One of the insts is a **taken** control flow inst
  - \[ nPC = \text{predicted target addr} \]
  - *NOTE* both instructions could be control-flow; prediction based on the first one predicted taken
  - If the 1\textsuperscript{st} instruction is the predicted taken branch
    \[ \Rightarrow \text{nullify 2\textsuperscript{nd} instruction fetched} \]
Multiple Instruction Fetch: Concepts

- Fetch 1 instr/cycle
- Downside: Flynn's bottleneck
  - If you fetch 1 instr/cycle, you cannot finish >1 instr/cycle.

- Fetch 4 instr/cycle

Two major approaches

1) VLIW
   - Compiler decides what runs.
   - Can be executed in parallel.
   - Simple hardware

2) Superscalar
   - Hardware detects dependencies between instructions that are fetched in the same cycle.
Review of Last Few Lectures

- Control dependence handling in pipelined machines
  - Delayed branching
  - Fine-grained multithreading
  - Branch prediction
    - Compile time (static)
      - Always NT, Always T, Backward T Forward NT, Profile based
    - Run time (dynamic)
      - Last time predictor
      - Hysteresis: 2BC predictor
      - Global branch correlation → Two-level global predictor
      - Local branch correlation → Two-level local predictor
  - Predicated execution
  - Multipath execution