Required Readings

- This week


- H&H Chapters 7.8 and 7.9

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Execution Paradigms
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays
Control Dependence Handling
Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?

- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction

- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?

- In fact, how do we even know whether or not the fetched instruction is a control-flow instruction?
### 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
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*)
Stall Fetch Until Next PC is Known: Good Idea?

This is the case with non-control-flow and unconditional br instructions!
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

- If we are fetching W instructions per cycle (i.e., if the pipeline is W wide)
  - A branch misprediction leads to $N \times W$ wasted instruction slots
Importance of The Branch Problem

- Assume $N = 20$ (20 pipe stages), $W = 5$ (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

How long does it take to fetch 500 instructions?

- **100% accuracy**
  - 100 cycles (all instructions fetched on the correct path)
  - No wasted work; IPC = $500/100$

- **99% accuracy**
  - 100 (correct path) + 20 * 1 (wrong path) = 120 cycles
  - 20% extra instructions fetched; IPC = $500/120$

- **90% accuracy**
  - 100 (correct path) + 20 * 10 (wrong path) = 300 cycles
  - 200% extra instructions fetched; IPC = $500/300$

- **60% accuracy**
  - 100 (correct path) + 20 * 40 (wrong path) = 900 cycles
  - 800% extra instructions fetched; IPC = $500/900$
Branch Prediction
Branch Prediction: Guess the Next Instruction to Fetch

![Diagram showing branch prediction and pipeline stages.

- PC
- I-$
- DEC
- RF
- D-$
- WB

Instructions:
1. LD R1, MEM[R0]
2. ADD R2, R2, #1
3. BRzero 0x0001
4. ADD R3, R2, #1
5. MUL R1, R2, R3
6. LD R2, MEM[R2]
7. LD R0, MEM[R2]

Cycles:
- Stall fetch: 12 cycles
- Branch prediction: 8 cycles
Misprediction Penalty

LD R0, MEM[R2]
LD R2, MEM[R2]
BR
ZERO
0x0001
MISPREDICTION PENALTY
LD R1, MEM[R0]
ADD R2, R2, #1
ADD R3, R2, #1
MUL R1, R2, R3
LD R2, MEM[R2]
LD R0, MEM[R2]

PC
I-$
DEC
RF
WB
Flush!!

0x0001
0x0002
0x0003
0x0004
0x0005
0x0006
0x0007

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]
Simplest: Always Guess \( \text{NextPC} = \text{PC} + 4 \)

- Always predict the next sequential instruction is the next instruction to be executed.
- This is a form of next fetch address prediction (and branch prediction).

- How can you make this more effective?

- Idea: Maximize the chances that the next sequential instruction is the next instruction to be executed.
  - Software: Lay out the control flow graph such that the “likely next instruction” is on the not-taken path of a branch.
    - Profile guided code positioning → Pettis & Hansen, PLDI 1990.
  - Hardware: ??? (how can you do this in hardware...)
    - Cache traces of executed instructions → Trace cache.
Guessing $\text{NextPC} = \text{PC} + 4$

- How else can you make this more effective?

- Idea: Get rid of control flow instructions (or minimize their occurrence)

- How?
  1. Get rid of unnecessary control flow instructions $\rightarrow$ combine predicates (predicate combining)
  2. Convert control dependences into data dependences $\rightarrow$ predicated execution
Branch Prediction: Always PC+4

When a branch resolves
- branch target (Inst\textsubscript{k}) is fetched
- all instructions fetched since inst\textsubscript{h} (so called “wrong-path” instructions) must be flushed
Pipeline Flush on a Misprediction

$\text{Inst}_h$ is a branch
Performance Analysis

- correct guess ⇒ no penalty  
  \( \sim 86\% \) of the time
- incorrect guess ⇒ 2 bubbles

Assume

- no data dependency related stalls
- 20% control flow instructions
- 70% of control flow instructions are taken

\[
\text{CPI} = \left[ 1 + (0.20 \times 0.7) \times 2 \right] = \\
= \left[ 1 + 0.14 \times 2 \right] = 1.28
\]

probability of penalty for a wrong guess

Can we reduce either of the two penalty terms?
Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early

Is this a good idea?

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

Program Counter
Address of the current branch

Direction predictor (taken?)

PC + inst size

Next Fetch Address

Cache of Target Addresses (BTB: Branch Target Buffer)
More Sophisticated Branch Direction Prediction

- Global branch history
- Program Counter
- Which direction earlier branches went
- Address of the current branch

Direction predictor (taken?)

XOR

PC + inst size

hit?

taken?

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address
Three Things to Be Predicted

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

- Third (3.) can be accomplished using a BTB
  - Remember target address computed last time branch was executed

- First (1.) can be accomplished using a BTB
  - If BTB provides a target address for the program counter, then it must be a branch
  - Or, we can store “branch metadata” bits in instruction cache/memory → partially decoded instruction stored in I-cache

- Second (2.): How do we predict the direction?
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
  - Advanced algorithms (e.g., using perceptrons)
Static Branch Prediction (I)

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

- **Always taken**
  - No direction prediction
  - Better accuracy: ~60-70% (for conditional branches)
    - 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 a profile run. Encodes that direction as a hint bit in the branch instruction format.

+ Per branch prediction (more accurate than schemes in previous slide) → accurate if profile is representative!
- Requires hint bits in the branch instruction format
- Accuracy depends on dynamic branch behavior:
  - TTTTTTTTTTNNNNNNNNNNN → 50% accuracy
  - TNTNTNTNTNTNTNTNTNTNTNTNTN → 50% accuracy
- Accuracy depends on the representativeness of profile input set
Static Branch Prediction (III)

- Program-based (or, program analysis based)
  - Idea: Use heuristics based on program analysis to determine statically-predicted direction
  - Example opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
  - Example 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 (ditto for other static methods)

  - 20% misprediction rate
Static Branch Prediction (IV)

- **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?
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 is the common disadvantage 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...)
  - What is a Dynamic Compiler?
    - A compiler that generates code at runtime
    - Java JIT (just in time) compiler, Microsoft CLR (common lang. runtime)
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
  - Advanced algorithms (e.g., using perceptrons)
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
    TTTTTTTTTTTNNNNNNNNNNN → 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 N (number of iterations)
-- Loop branches for loops will small N (number of iterations)
  TNTNTNTNTNTNTNTNTNTNTNTNTN → 0% accuracy
We did not cover the following slides. They are for your preparation for the next lecture.
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

- **Predict not taken**
  - Transition to **Actually not taken**
  - Transition to **Predict taken**

- **Predict taken**
  - Transition to **Actually not taken**
  - Transition to **Actually taken**

**Transitions:***
- Actually taken → Predict taken
- Actually not taken → Predict not taken
- Predict not taken → Actually not taken
- Predict taken → Actually not taken
Improving the Last Time Predictor

- **Problem:** A last-time predictor changes its prediction from T→NT or NT→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 = \( \frac{(N-1)}{N} \)
  TNTNTNTNTNTNTNTNTNTNTNTN → 50% accuracy
  (assuming counter initialized to weakly taken)

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

- Counter using **saturating arithmetic**
  - Arithmetic with maximum and minimum values

![State Machine Diagram]

- actually taken
- actually !taken

pred taken 11

pred taken 10

pred !taken 01

pred !taken 00

actually taken

actually !taken

actually taken

actually !taken

actually taken

actually !taken
Hysteresis Using a 2-bit Counter

Change prediction after 2 consecutive mistakes
Is This Good 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?
Let’s Do the Exercise Again

- Assume $N = 20$ (20 pipe stages), $W = 5$ (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

How long does it take to fetch 500 instructions?

- **100% accuracy**
  - 100 cycles (all instructions fetched on the correct path)
  - No wasted work; IPC = $500/100$

- **90% accuracy**
  - 100 (correct path) + 20 * 10 (wrong path) = 300 cycles
  - 200% extra instructions fetched; IPC = $500/300$

- **85% accuracy**
  - 100 (correct path) + 20 * 15 (wrong path) = 400 cycles
  - 300% extra instructions fetched; IPC = $500/400$

- **80% accuracy**
  - 100 (correct path) + 20 * 20 (wrong path) = 500 cycles
  - 400% extra instructions fetched; IPC = $500/500$
Can We Do Better: Two-Level 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

Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path are correlated with the outcome of the next branch
  
  ```
  if (cond1)
  ...
  if (cond1 AND cond2)
  ```

- If first branch not taken, second also not taken
  
  ```
  branch Y: if (cond1) a = 2;
  ...
  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’92: Generates truth table from Boolean expr.

```c
if (aa==2) 
    aa=0;
if (bb==2) 
    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 that recorded the outcome that was seen for each 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

How Does the Global Predictor Work?


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></td>
<td>1110</td>
<td>usually taken</td>
</tr>
</tbody>
</table>

This branch tests i
Last 4 branches test j
History: TTTN
Predict taken for i
Next history: TTNT
(shift in last outcome)
Intel Pentium Pro Branch Predictor

- Two level global 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 used for prediction
  - Better utilization of the two-bit counter array
  -- Increases access latency

Review: One-Level Branch Predictor

Direction predictor (2-bit counters)

Program Counter

Address of the current instruction

Cache of Target Addresses (BTB: Branch Target Buffer)

PC + inst size

taken?

hit?

target address

Next Fetch Address
Two-Level Global History Branch Predictor

- **Global branch history**
- **Program Counter**
- Which direction earlier branches went
- Cache of Target Addresses (BTB: Branch Target Buffer)
- Take? or hit?
- PC + inst size
- Next Fetch Address
- Target address
- Direction predictor (2-bit counters)
Two-Level Gshare Branch Predictor

- Global branch history
- Program Counter
- XOR
- Which direction earlier branches went
- Address of the current instruction
- Direction predictor (2-bit counters)
- Taken?
- Hit?
- PC + inst size
- Next Fetch Address
- Cache of Target Addresses (BTB: Branch Target Buffer)
Can We Do Better: Two-Level Prediction

- Last-time and 2BC predictors exploit only “last-time” predictability for a given branch

- 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 (in addition to the outcome of the branch “last-time” it was executed)
  - 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.

More Motivation for Local History

- To predict a loop branch "perfectly", we want to identify the last iteration of the loop.

- By having a separate PHT entry for each local history, we can distinguish different iterations of a loop.

- Works for "short" loops.

Loop closing branch’s history

```
1110 1110 1110 1110 1110 1110 1110 1110
```

PHT
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 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 Branch 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)

Next Fetch Address

PC + inst size

hit?

taken?
Can We Do Even Better?

- Predictability of branches varies
- Some branches are more predictable using local history
- Some using global
- For others, a simple two-bit counter is enough
- Yet for others, a single bit is enough

Observation: There is heterogeneity in predictability behavior of branches
  - No one-size fits all branch prediction algorithm for all branches

Idea: Exploit that heterogeneity by designing heterogeneous branch predictors
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

Are We Done w/ Branch Prediction?

- Hybrid branch predictors work well
  - E.g., 90-97% prediction accuracy on average

- Some “difficult” workloads still suffer, though!
  - E.g., gcc
  - Max IPC with tournament prediction: 9
  - Max IPC with perfect prediction: 35
Some Other Branch Predictor Types

- Loop branch detector and predictor
  - Loop iteration count detector/predictor
  - Works well for loops with small number of iterations, where iteration count is predictable
  - Used in Intel Pentium M

- Perceptron branch predictor
  - Learns the *direction correlations* between individual branches
  - Assigns weights to correlations

- Hybrid history length based predictor
  - Uses different tables with different history lengths
The advanced branch prediction in the Pentium M processor is based on the Intel Pentium® 4 processor’s [6] branch predictor. On top of that, two additional predictors to capture special program flows, were added: a Loop Detector and an Indirect Branch Predictor.

Figure 2: The Loop Detector logic

Figure 3: The Indirect Branch Predictor logic

Perceptrons for Learning Linear Functions

- A perceptron is a simplified model of a biological neuron
- It is also a simple **binary classifier**

- A perceptron maps an input vector $X$ to a 0 or 1
  - Input = Vector $X$
  - Perceptron learns the linear function (if one exists) of how each element of the vector affects the output (stored in an internal Weight vector)
  - Output = Weight.$X + Bias > 0$

- In the branch prediction context
  - Vector $X$: Branch history register bits
  - Output: Prediction for the current branch
Perceptron Branch Predictor (I)

- **Idea:** Use a perceptron to learn the correlations between branch history register bits and branch outcome
- **A perceptron learns a target Boolean function of N inputs**

Each branch associated with a perceptron

A perceptron contains a set of weights $w_i$

- Each weight corresponds to a bit in the GHR
- How much the bit is correlated with the direction of the branch
- Positive correlation: large + weight
- Negative correlation: large - weight

Prediction:
- Express GHR bits as 1 (T) and -1 (NT)
- Take dot product of GHR and weights
- If output > 0, predict taken

Perceptron Branch Predictor (II)

Prediction function:

\[ y = w_0 + \sum_{i=1}^{n} x_i w_i. \]

Output compared to 0

Bias weight (bias of branch, independent of the history)

Bias weight

Training function:

\[
\begin{align*}
\text{if } \text{sign}(y_{out}) \neq t \text{ or } |y_{out}| \leq \theta \text{ then } \\
\text{for } i := 0 \text{ to } n \text{ do } \\
\quad w_i := w_i + tx_i \\
\text{end for } \\
\text{end if }
\end{align*}
\]
Perceptron Branch Predictor (III)

- **Advantages**
  + More sophisticated learning mechanism → better accuracy

- **Disadvantages**
  -- Hard to implement (adder tree to compute perceptron output)
  -- Can learn only linearly-separable functions
    e.g., cannot learn XOR type of correlation between 2 history bits and branch outcome
Observation: Different branches require different history lengths for better prediction accuracy.

Idea: Have multiple PHTs indexed with GHRs with different history lengths and intelligently allocate PHT entries to different branches.

Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths.

State of the Art in Branch Prediction

- See the Branch Prediction Championship
  - [https://www.jilp.org/cbp2016/program.html](https://www.jilp.org/cbp2016/program.html)

Andre Seznec, 

Andre Seznec, 

Figure 1. The TAGE-SC-L predictor: a TAGE predictor backed with a Statistical Corrector predictor and a loop predictor
Branch Confidence Estimation

- **Idea:** Estimate if the prediction is likely to be correct
  - i.e., estimate how “confident” you are in the prediction

- **Why?**
  - Could be very useful in deciding how to speculate:
    - What predictor/PHT to choose/use
    - Whether to keep fetching on this path
    - Whether to switch to some other way of handling the branch, e.g. dual-path execution (eager execution) or dynamic predication
    - ...

Other Ways of Handling Branches
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:

\[
\begin{align*}
A & \\
B & \\
C & \\
BC & X \\
D & \\
E & \\
F & \\
G & \\
\end{align*}
\]

\[\text{X:} \ \begin{array}{l} X: \end{array}\]

Timeline:

\[
\begin{align*}
\text{if} & \quad \text{ex} \\
A & \\
B & \ A \\
C & \ B \\
BC & \ C \\
-- & \ BC \\
G & \ -- \\
\end{align*}
\]

\[\text{6 cycles}\]

Delayed branch code:

\[
\begin{align*}
A & \\
C & \\
BC & X \\
B & \\
D & \\
E & \\
F & \\
G & \\
\end{align*}
\]

\[\text{X:} \ \begin{array}{l} X: \end{array}\]

Timeline:

\[
\begin{align*}
\text{if} & \quad \text{ex} \\
A & \\
C & \ A \\
BC & \ C \\
B & \ BC \\
G & \ B \\
\end{align*}
\]

\[\text{5 cycles}\]
Fancy Delayed Branching (III)

- Delayed branch with squashing
  - In SPARC
  - Semantics: If the branch falls through (i.e., it is 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 the same basic block:

- **For correctness:** add a new instruction to the not-taken path?
- **For correctness:** add a new instruction to the taken path?

For correctness:

- Add a new instruction to the not-taken path?
- Add a new instruction to the taken path?
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**)
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 instead of having one branch for each
  - 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:** Convert control dependence to data dependence

- **Simple example:** 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;
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)

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

(predicated code)

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

```plaintext
A
  T
  C
  B
  D
```

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

```
p1 = (cond)
(!p1) mov b, 1
(p1) mov b, 0
add   x, b, 1
```
Predicated Execution References


Conditional Move Operations

- Very limited form of predicated execution

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

- Predicated execution can be high performance and energy-efficient
Predicated Execution

- Eliminates branches → enables straight line code (i.e., larger basic blocks in code)

**Advantages**
- Eliminates hard-to-predict branches
- Always-not-taken prediction works better (no branches)
- Compiler has more freedom to optimize code (no branches)
  - control flow does not hinder inst. reordering optimizations
  - code optimizations hindered only by data dependencies

**Disadvantages**
- Useless work: some instructions fetched/executed but discarded (especially bad for easy-to-predict branches)
- Requires additional ISA (and hardware) support
- Can we eliminate all branches this way?
**Predicated Execution vs. Branch Prediction**

+ Eliminates mispredictions for hard-to-predict branches
  + No need for branch prediction for some branches
  + Good if misprediction cost > useless work due to predication

-- 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, program phase, control-flow path.
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
Almost all ARM instructions can include an optional condition code.

Prior to ARM v8

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Opcode</th>
<th>Immediate</th>
<th>Register</th>
<th>Immediate</th>
<th>Condition</th>
<th>Immediate</th>
<th>Register List</th>
<th>Immediate</th>
<th>Condition</th>
<th>Immediate</th>
<th>Register</th>
<th>Immediate</th>
<th>Condition</th>
<th>Immediate</th>
<th>Register List</th>
<th>Immediate</th>
<th>Condition</th>
<th>Immediate</th>
<th>Register</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 1 0 1 0 0 1 0 1 0 1 0 1 1 1 1</td>
<td>11 0 0 1 0 1 0 1 0</td>
<td>11 1 1 1 1 1 1 1 0 0 1</td>
<td>Rd</td>
<td>CRd</td>
<td>CPNum</td>
<td>Offset</td>
<td>1 1 1 0 OpL</td>
<td>CRn</td>
<td>CRd</td>
<td>CPNum</td>
<td>Op2</td>
<td>0</td>
<td>CRm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0 OpL</td>
<td>CRn</td>
<td>CRd</td>
<td>CPNum</td>
<td>Op2</td>
<td>0</td>
<td>CRm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0 OpL</td>
<td>CRn</td>
<td>CRd</td>
<td>CPNum</td>
<td>Op2</td>
<td>1</td>
<td>CRm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>SWI Number</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conditional Execution in ARM ISA

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

```
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?
```
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 context (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

path 1

path 2

CFMerge

CFMerge
Handling Other Types of Branches
### 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>

How can we predict an indirect branch with many target addresses?
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

![Diagram of indirect branch prediction](chart)

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

```
R1 = MEM[R2]
branch R1
```
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
The advanced branch prediction in the Pentium M processor is based on the Intel Pentium® 4 processor’s [6] branch predictor. On top of that, two additional predictors to capture special program flows, were added: a Loop Detector and an Indirect Branch Predictor.

Figure 2: The Loop Detector logic

Figure 3: The Indirect Branch Predictor logic

Gochman et al.,
“The Intel Pentium M Processor: Microarchitecture and Performance,”
Issues in Branch Prediction (I)

- Need to identify a branch before it is fetched

- How do we do this?
  - BTB hit $\rightarrow$ indicates that the fetched instruction is a branch
  - BTB entry contains the “type” of the branch
  - Pre-decoded “branch type” information stored in the instruction cache identifies type of branch

- What if no BTB?
  - Bubble in the pipeline until target address is computed
  - E.g., IBM POWER4
Latency of Branch Prediction

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