Branches

Introduction

1. Performance will not be good unless we address the fact that control hazards waste many cycles

Branch in a Pipeline

1. BEQ R1, R2, Label
   - R1 != R2 -> PC++
   - R1 == R2 -> PC = PC + 4 + Immediate
2. Regardless of whether or not the branch is taken, we've fetched the next two instructions
   - In the ALU stage, the target of the branch is determined
   - If the branch is not taken, this is correct
   - If it is taken, we need to cancel the previous two instructions
3. Mispredicting the branch causes the branch to effectively take 3 cycles to execute instead of 1
   - Cost of misprediction is 2 cycles
   - Should still try though, leaving a bubble will always cause a 2 cycle penalty

Branch Prediction Requirements

1. Know only of PC of instruction
   - Guess PC of next instruction to fetch
2. Must correctly guess
   - Is this a branch?
   - Is it taken?
   - (Is this a taken branch?)
   - If taken, what is the target PC?

Branch Prediction Accuracy

1. CPI = 1 + (MISPREDS/INST) * (PENALTY/MISPRED)
   - MISPREDS/INST -> Predictor accuracy
   - PENALTY/MISPRED -> Pipeline
2. Assume 20% of all instructions are branches
3. Accuracy = 50% for branch, 100% for all other instructions
   - Resolve branch in 3rd stage: 1 + 0.5 * 0.2 * 2 = 1.2
   - Resolve branch in 10th stage: 1 + 0.5 * 0.2 * 9 = 1.9
4. Accuracy = 90% for branch, 100% for all other instructions
   - Resolve branch in 3rd stage: 1 + 0.1 * 0.2 * 2 = 1.04
   - Resolve branch in 10th stage: 1 + 0.1 * 0.2 * 9 = 1.18
5. A better branch predictor will help regardless of pipeline depth
   - Amount of help depends on depth of pipeline
   - Speedup = 1.15 for shallow pipeline
   - Speedup = 1.61 for deep pipeline
6. Modern branch predictors are significantly better than 90% accurate
   - Modern pipelines are deep, so we benefit from further improvements

Branch Prediction Benefit Quiz

1. 5-stage pipeline
2. Branch resolved in 3rd stage
3. Fetch nothing until sure what to fetch
4. Execute many iterations of:

```
LOOP: ADDI R1, R1, -1
      ADD R2, R2, R2
      BNE R1, LOOP
```

5. Speedup of predictor is: 7/3 = 2.33
   - ADD and ADDI costs 2 cycles (since we don't know if the instruction is a branch until the decode stage, we don't fetch the next instruction)
   - BNEZ costs 3 cycles since we don't know its target until the ALU stage
   - 7 cycles per iteration with no predictor
   - 3 cycles per iteration with perfect predictor

Performance with Not Taken Prediction

1. Refusing to predict
   - Branch costs 3 cycles
   - Non-branch costs 2 cycles
2. Predict not-taken
   - Branch: 1 or 3 cycles
   - Non-branch: 1 cycle
3. Predict not-taken is simple, only have to increment the PC

Multiple Predictions Quiz

1. Consider the following program:

```
BNE R1, R2, Label A (taken)
BNE R1, R3, Label B (taken)
A
B
C
Label A: X
Y
Label B: Z
```

2. How many cycles are wasted on mispredictions until we get to Y?
   - Have fetched second branch and A, resulting in 2 cycles wasted
   - Second mis-prediction causes no further penalty

Predict Not Taken

1. Simply increment PC (no memory)
2. 20% of instructions are branches, 60% of branches are taken
   - Correct: 80% + 8% = 88%
   - Incorrect: 12%
   - CPI = 1 + 0.12 * PENALTY = 1.24

Why We Need Better Prediction

1. Better branch predictor improves CPI

Penalty	Not Taken (88%)	Better (99%)	Speedup
5-Stage (3rd)	1+0.12*2 = 1.24	1+0.01*2 = 1.02	1.22
14-Stage (11th)	1+0.12*10 = 2.2	1+0.01*10 = 1.1	2
14-Stage 4 inst/cycle	0.25 + 0.12*10 = 1.45	0.25+0.01*10 = 0.35	414

Predictor Impact Quiz

1. Pentium 4 "Prescott"
   - Fetch, 29 stages, resolve branches
   - Branch prediction
   - Multiple instructions/cycle
2. Program
   - 20% of instructions are branches
   - 1% of branches mispredicted
   - CPI = 0.5 = X + 0.2 * 0.01 * 30 -> X = 0.5 - 0.06 = 0.44
3. If 2% of branches are mispredicted, CPI = 0.44 + 0.2 * 0.02 * 30 = 0.56

Why We Need Better Prediction 2

1. Mis-prediction causes instructions to be wasted
   - 5-stage pipeline: 2 wasted instructions
   - 14-stage pipeline: 10 wasted instructions
   - 14-stage pipeline with 4 inst/cycle: 40 wasted instructions

Better Prediction - How?

1. PCnext = f(PCnow)
   - Is this a branch?
   - Is it taken?
   - What's the offset?
2. PCnext = f(PCnow, History[PCnow])
   - Branches tend to behave in the same way

Branch Target Buffer (BTB)

1. PCnow indexes into a table (BTB)
   - Table contains our best guess for next PC
   - Once true target is determined, compare to predicted
     - At this point, index into BTB with PC and update its value
2. How big should the BTB be?
   - 1 cycle latency means BTB should be small
   - 64 bit addressing -> 8-byte entry
   - 1 entry/PC -> this will result in a very large BTB
3. How do we reconcile the need for a large BTB with small latency?

Realistic BTB

1. Don't need an entry for every possible PC, just the instructions that are likely to execute soon
   - Only 1024 entries can be accessed in one cycle
   - How do we map PC onto BTB?
     - Need 10 bits to address our BTB (2^10 = 1024)
     - Use the lowest 10 bits of the PC

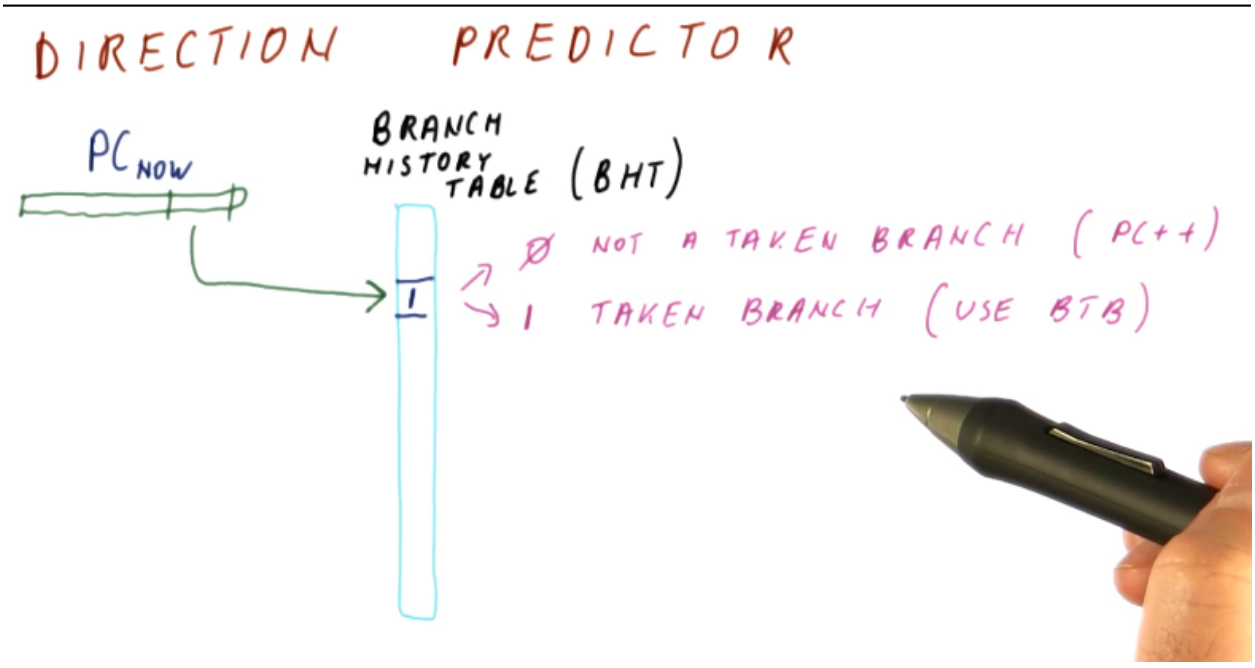
BTB Quiz

1. BTB with 1024 entries (0..1023)
2. Fixed-size 4-byte instructions, word-aligned
3. 32-bit address
4. Which BTB entry is used for PC = 0x0000AB0C?
   - Only valid entries in BTB are multiples of 4, so we should use bits 2-11 instead of 0-9

- 1011 0000 1011 -> 1011000011 -> 0x2C3

Direction Predictor

1. Branch History Table (BHT)
   - Simplest predictor is a single bit (0: Not taken, 1: Taken)
     - 0: Not a taken branch (PC++)
     - 1: Taken branch (use BTB)
   - Table can be large since an entry is only 1 bit



Direction Predictor

BTB and BHT Quiz 1

1. Consider the following program:

```
0xC000       MOV R2, 100
0xC004       MOV R1, 0
0xC008 Loop: BEQ R1, R2, Done
0xC00C       ADD R4, R3, R1
0xC010       LW R4, 0(R4)
0xC014       ADD R5, R5, R4
0xC018       ADD R1, R1, 1
0xC01C       B Loop
0xC020 Done:
```

2. BHT with 16 entries, perfect predictions
3. BTB with 4 entries, perfect predictions
4. How many times do we access the BHT for each instruction?

```
0xC000       MOV R2, 100 (1)
0xC004       MOV R1, 0  (1)
0xC008 Loop: BEQ R1, R2, Done (101)
0xC00C       ADD R4, R3, R1 (100)
```

```
0xC010       LW R4, 0(R4) (100)
0xC014       ADD R5, R5, R4 (100)
0xC018       ADD R1, R1, 1 (100)
0xC01C       B Loop (100)
0xC020 Done:
```

BTB and BHT Quiz 2

1. Which BHT entry do we access for each instruction?

```
0xC000       MOV R2, 100 (0)
0xC004       MOV R1, 0 (1)
0xC008 Loop: BEQ R1, R2, Done (2)
0xC00C       ADD R4, R3, R1 (3)
0xC010       LW R4, 0(R4) (4)
0xC014       ADD R5, R5, R4 (5)
0xC018       ADD R1, R1, 1 (6)
0xC01C       B Loop (7)
0xC020 Done:
```

BTB and BHT Quiz 3

1. How many times do we access the BTB for each instruction?

```
0xC000       MOV R2, 100 (0)
0xC004       MOV R1, 0  (0)
0xC008 Loop: BEQ R1, R2, Done (1)
0xC00C       ADD R4, R3, R1 (0)
0xC010       LW R4, 0(R4) (0)
0xC014       ADD R5, R5, R4 (0)
0xC018       ADD R1, R1, 1 (0)
0xC01C       B Loop (100)
0xC020 Done:
```

BTB and BHT Quiz 4

1. Which BTB entry do we use for each instruction?

```
0xC000       MOV R2, 100
0xC004       MOV R1, 0
0xC008 Loop: BEQ R1, R2, Done (2)
0xC00C       ADD R4, R3, R1
0xC010       LW R4, 0(R4)
0xC014       ADD R5, R5, R4
0xC018       ADD R1, R1, 1
0xC01C       B Loop (4)
0xC020 Done:
```

BTB and BHT Quiz 5

1. BHT: 16 entries, 1-bit initialized to not taken
2. How many mispredictions do we have for each instruction?

```
0xC000       MOV R2, 100 (0)
0xC004       MOV R1, 0 (0)
0xC008 Loop: BEQ R1, R2, Done (1)
```

```
0xC00C       ADD R4, R3, R1 (0)
0xC010       LW R4, 0(R4) (0)
0xC014       ADD R5, R5, R4 (0)
0xC018       ADD R1, R1, 1 (0)
0xC01C       B Loop (1)
0xC020 Done:
```

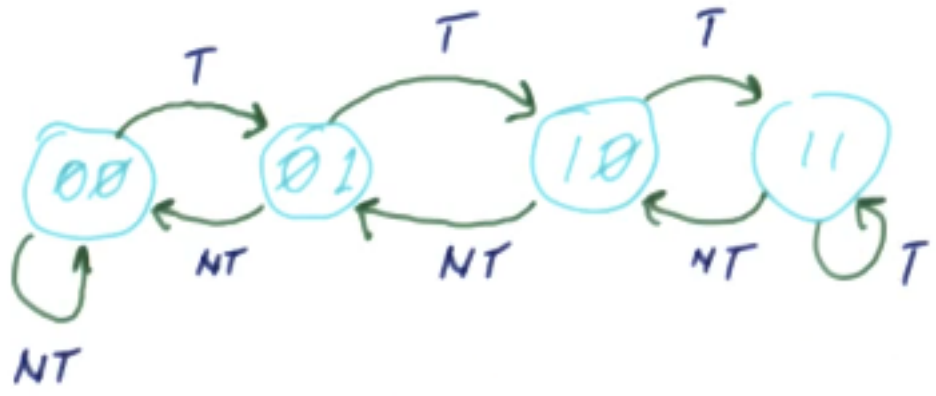
3. 1-bit predictor works well for loops with many iterations

Problems with 1 Bit Predictions

1. Predicts well:
   - Branches that are always taken
   - Branches that are always not taken
   - Branches where taken >> not taken
   - Branches where not taken >> taken
2. Doesn't predict well:
   - T > NT
   - NT > T
   - Short loop
   - T is similar to NT
3. 1-bit predictors break down in the anomalous case T -> NT -> T
   - There are two missed predictions in this case

2 Bit Predictor (2BP, 2BC)

1. 2BC: 2-bit counter
   - First bit is prediction bit
   - Second bit is hysteresis (conviction) bit
   - 00: Strong not taken
   - 01: Weak not taken
   - 10: Weak taken
   - 11: Strong taken
2. Counter that increases when branch is taken and decreases when not taken
   - Saturates at 0 and 3



2-Bit Predictor State Machine

2 Bit Predictor Initialization

1. Start in strong state

- Correct strong state: No mispredictions
- Incorrect strong state: 2 mispredictions

2. Start in weak state
   - Correct weak state: No mispredictions
   - Incorrect weak state: 1 misprediction
3. If the branch alternates between taken and not taken...
   - Starting in a strong state results in every other prediction being wrong
   - Starting in a weak state results in every prediction being wrong
4. Taken branches are slightly more common than not taken, so it makes sense to start in a weak taken state, in theory (alternating is not the common case)
   - In practice, overall accuracy of branch prediction isn't affected by one more correct prediction, so might as well start in 00 for ease

2 Bit Predictor Quiz

1. Assume a typical 2-bit branch predictor
2. Start in strong NT state
3. Is there a sequence of branch outcomes that results in never predicting correctly?
   - Yes; T -> T -> NT -> T -> N
   - Every predictor has a worst-case behavior resulting in 100% mispredictions
     - Goal is to minimize the likelihood of this case

1BP -> 2BP

1. Is there an additional advantage to using a 3-bit or 4-bit predictor?
   - Pros: When "anomalous" outcomes come in streaks
     - How often does this happen in real programs?
   - Cons: Cost increases
2. Stay with 2BP, potentially use 3BP
   - How do we further improve prediction?
     - T -> NT -> T -> NT -> T is perfectly predictable, but not with 1BP or 2BP

History Based Predictors

1. Goal is to predict patterns correctly
   - N T N T N T ...
   - N N T N N T ...
   - 100% predictable, but not with simple n-bit counter
2. Learn the pattern
   - N T N T N T ...
     - N -> T, T -> N
   - N N T N N T ...
     - NN -> T
     - NT -> N
     - TN -> N

1 Bit History with 2 Bit Counters

1. Same approach; bits from PC used to index into BHT
   - Instead of a 1-bit or 2-bit counter, we have a single bit for history and 2 2-bit counters
2. In each state, use the history from the previous to update the counter

State	Pred	Outcome	Correct?
(0,SN,SN)	N	T	N

State	Pred	Outcome	Correct?
(1,WN,SN)	N	N	Y
(0,WN,SN)	N	T	N
(1,WT,SN)	N	N	Y
(0,WT,SN)	T	T	Y
(1,ST,SN)	N	N	Y
(0,ST,SN)	T	T	Y

1 Bit History Quiz

1. 1-bit history (start 0)
2. 2-bit counter / history (start SN)
3. NNTNNTNNT...
4. After 100 repetitions, the number of mispredictions is 100

State	Pred	Outcome	Correct?
(0,SN,SN)	N	N	Y
(0,SN,SN)	N	N	Y
(0,SN,SN)	N	T	N
(1,WN,SN)	N	N	Y
(0,WN,SN)	N	N	Y
(0,SN,SN)	N	T	N
(1,WN,SN)	N	N	Y
(0,WN,SN)	N	N	Y
(0,SN,SN)	N	T	N

2 Bit History Predictor

1. 1-bit history is insufficient for this pattern (NNTNNTNNT)
   - 2-bit history resolves this
2. 2-bit history: 2 bits of history with 4 (2 ^ 2) 2-bit counters
   - 10 bits per branch
3. History of NN corresponds to counter 0
   - This counter will be incremented, so it will soon begin predicting ST
4. History of NT corresponds to counter 1
   - This counter will be decremented, so it will continue predicting SN
5. History of TN corresponds to counter 2
   - This counter will be decremented, so it will continue predicting SN

2 Bit History Predictor 2

1. After the warmup period NNT... -> 100%
   - Wastes 1 2BC
2. How does a 2-bit counter perform on NTNTNTNT?
   - NT -> Counter 1, continually decremented
   - TN -> Counter 2, continually incremented
   - This will predict correctly, but waste 2 2BC
3. N-bit history predictor will correctly predict all patterns of length less than or equal to N+1
   - Cost is N + 2 * (2 ^ N)
   - Most of the 2BCs will be wasted
     - For a 10 bit history, can predict sequences of length 11, but use 1024 bits
4. Want a long history for loops, but without large cost and waste

N Bit History Predictor Quiz

1. N-bit history, 2BC / history
2. Need 1024 entries

N bits	Cost (bits)	NNNT...	# 2BCs used for NT...
N = 1	5 * 2^10	N	2
N = 4	(4 + 2 * 2 ^ 4) * 2 ^ 10	Y	2
N = 8	532480	Y	2
N = 16	134234112	Y	2

History Predictor Quiz

1. Consider the following program:

```
for(int i = 0; i < 8; i++)
{
    for(int j = 0; j < 8; j++)
    {
        doSomething();
    }
}
```

2. To build a branch predictor for this code, we need at least 4 entries
   - Each entry should have at least 8-bit history
   - This implies 2^8 2-bit counters

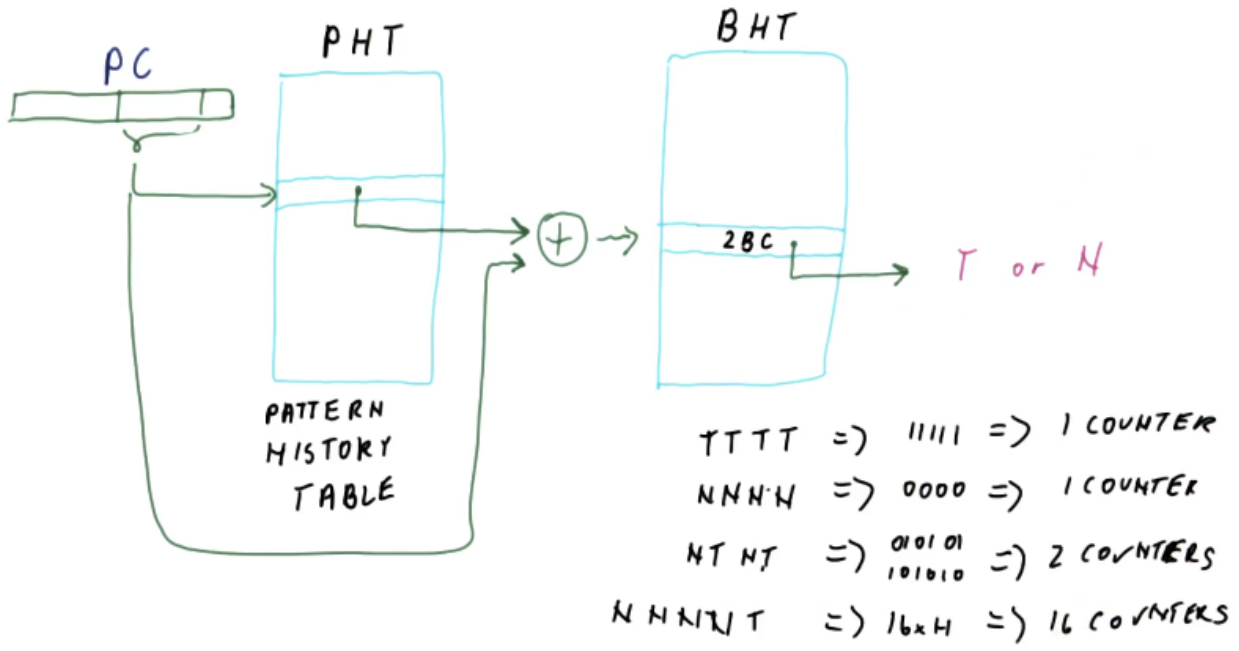
History with Shared Counters 1

1. Want to reduce waste but still predict long history
   - N-bit history -> 2^N counters per entry
   - Only ~ N counters used (proportional to size of history)
2. Idea: Share 2BCs between entries

History with Shared Counters 2

1. Use low bits of PC to index into Pattern History Table (PHT)
   - Take history from table, XOR with PC to index into BHT
   - BHT contains a single 2BC per entry
     - This indicates taken or not taken
     - Update this table depending on whether a branch is taken or not
   - Multiple PCs can map to the same entry in PHT, but when XOR'ed, should map to different entries in the BHT
2. 11-bits of history
   - PHT is 2^11 entries, each entry is 11 bits
   - Total cost: 2^11 for PHT + 2^11 * 2 for counters = 26kB
   - Entries in PHT and size of entry don't have to be the same

History with Shared Counters 3



History with Shared Counters

PShare

1. Pshare predictor: Private history for each branch with shared counters
   - Good for even-odd, 8-iteration loop
2. Gshare predictor: Global history for all branches with shared counters
   - Good for correlated branches, as follows:
   - if(shape == square) { ... }
   - ...
   - if(shape != square) { ... }

PShare vs GShare Quiz

1. Consider the following program:

```
for(int i = 1000; i > 0; i--)
{
    if(i % 2) { n += i; }
}
```

2. Maps to the following assembly:

```
Loop: BEQ R1, Zero, Exit
      AND R2, R1, 1
      BEQ R2, Zero, Even
      ADD R3, R3, R1
Even: ADD R1, R1, -1
      B Loop
Exit:
```

3. How many bits of history for good accuracy on all branches?
   - Pshare: 1
   - Gshare: 3

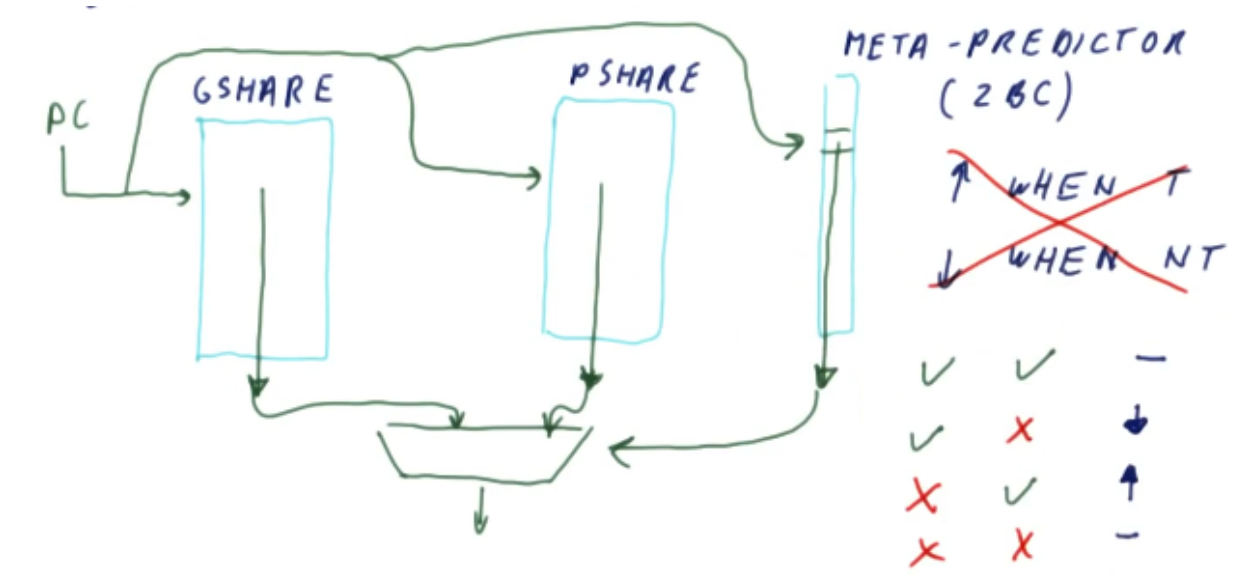
4. B Loop and BEQ R1, Zero, Exit are easy to predict
    - BEQ R2, Zero, Even alternates, so it needs 1 bit of history for Pshare
    - For Gshare, need 1 bit for each branch, so 3 total

GShare or PShare

1. Pshare pro: Requires less history
2. Gshare pro: Can handle correlated branches at the expense of more history bits
3. As a result, processors typically include both

Tournament Predictor

1. Two predictors
    - One better for branches A, B, C
    - One better for branches X, Y, Z
2. Use PC to index into both Pshare and Gshare and determine which to use
    - Meta-predictor: 2BC to select between two choices
3. Meta-predictor isn't trained on taken vs not taken, but on how well the two predictors are performing
    - If both predictors are right, do nothing
    - If first predictor is right, count down to select it more often
    - If second predictor is right, count up to select it more often
    - If both predictors are wrong, do nothing



Tournament Predictor

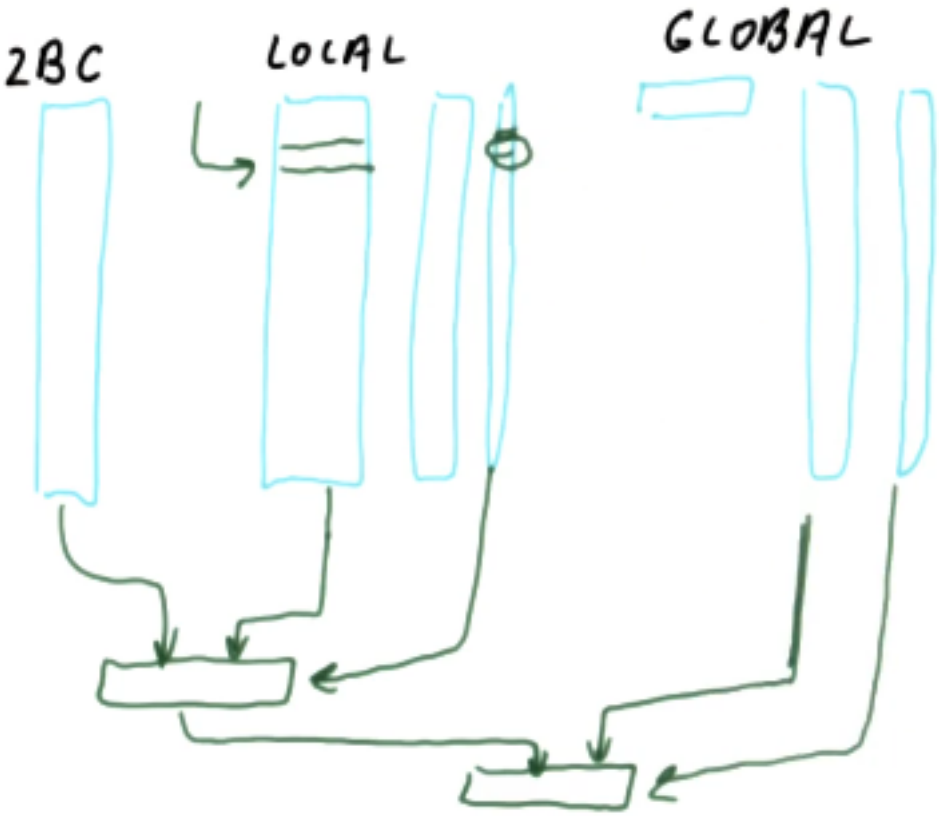
Hierarchical Predictors

1. Tournament predictor uses 2 good predictors
    - High cost to only use one of the answers
    - Update both predictors on each decision to keep them current
2. Hierarchical predictor uses 1 good predictor and 1 OK predictor
    - Don't use the expensive predictor for branches that are easy to predict
    - Update OK-predictor on each decision, but only update the good-predictor if the OK-predictor was incorrect

3. Hierarchical usually outperforms tournament because there are typically many branches that can be predicted using only a 2BC
    - Can make the good-predictor even better than a good predictor in a tournament predictor since we're saving resources with the OK-predictor

Hierarchical Predictor Example

1. Pentium M predictors
    - Cheap (2BC)
    - Local history
    - Global history
2. When making a prediction, look up 2BC, local, and global history
    - Form the actual prediction using the result of the global predictor if the branch is predicted
        - Global predictor has a tag to indicate if the prediction exists
    - If the global array doesn't predict it, check the tag for the local predictor
    - If the tag doesn't exist in the local predictor, go to the 2BC
3. Updating the predictors
    - If the 2BC mispredicts, update the local predictor
    - If the local predictor mispredicts, update the global predictor



Hierarchical Predictor

Multi Predictor Quiz

1. 2BP works fine for 95% of instructions
2. Pshare works fine for the same 95%, plus 2% more
3. Gshare works fine for the same 95%, plus the other 3% (different from Pshare)
4. The overall predictor is a hierarchical predictor that chooses between a 2BP and tournament predictor, which itself chooses between a Pshare and Gshare predictor

Return Address Stack (RAS)

1. BNE R0, R1, Label
   - Need to predict direction (as shown above)
   - Also need to predict target (BTB)
2. JUMP, CALL, etc.
   - Direction is trivial (always taken)
   - Target can be handled with BTB
3. Function return
   - Direction is trivial (always taken)
   - Target is easy if it's always called from the same location, but functions are intended to be called from many places
     - This complicates the return address target
     - BTB will predict the wrong place

RAS

1. Separate predictor for predicting function returns
2. When we have a function call, we push the address to the RAS
   - Then, on return, we simply pop the stack
   - Handles nested functions trivially
3. Don't want to use the call stack because we want this to be fast and close to the rest of the branch prediction logic
4. RAS is much smaller than call stack; what do we do if it's full?
   - Option 1: Don't push anything
   - Option 2: Wrap around and overwrite

RAS Full Quiz

1. Which is better; don't push or wrap around?
   - Wrap around is better
2. Wrap around strategy is preferable because there can be many calls and returns
   - If we stop pushing, all of this is lost to preserve the initial calls
   - By wrapping around, we can overwrite small function calls many times, but still predict the target correctly
3. Neither will do things perfectly, but wrap around performs better

But How Do We Know It's a RET?

1. Need to make prediction while fetching the instruction
   - Can't just push and pop if it isn't a CALL or RET, but hasn't been decoded yet
2. Option 1: Train a predictor to determine if an instruction is a RET
   - Very accurate
3. Option 2: Predecoding - Annotate instruction in a cache
   - Processor fetches from cache instead of memory, only goes to memory if the cache doesn't contain the instruction

- Cache can store 32-bits for instruction plus a extra bits for predecoding the instruction (i.e., flag for RET, flag for BRANCH)

4. Modern processors do a lot of work in predecode stage to save work in the actual pipeline

Conclusion

1. Some branches are very difficult to predict, so we can try to help the compiler eliminate these types of branches entirely