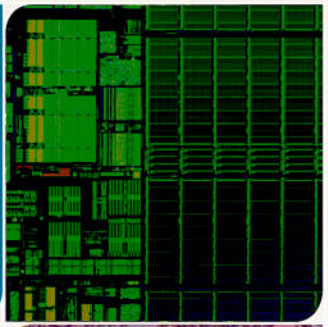
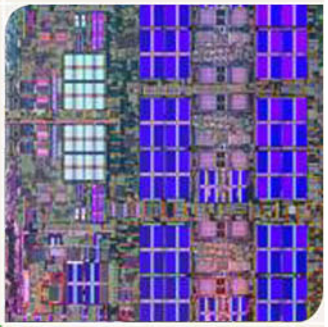
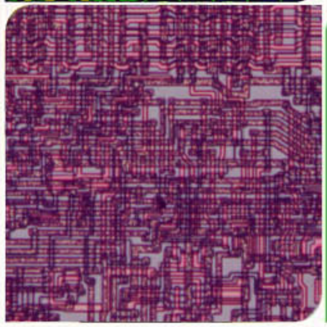
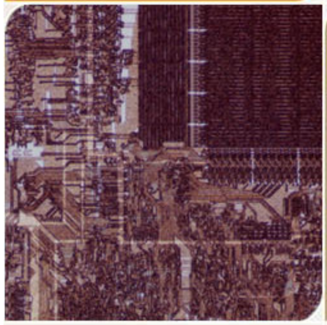





CS4803DGC Design Game Consoles

Spring 2009

Prof. Hyesoon Kim



Georgia Tech College of Computing

Control Dependencies

- Branches are very frequent
  - Approx. 20% of all instructions
- Can not wait until we know where it goes
  - Long pipelines
    - Branch outcome known after B cycles
    - No scheduling past the branch until outcome known
  - Superscalars (e.g., 4-way)
    - Branch every cycle or so!
    - One cycle of work, then bubbles for ~B cycles?

What to do with branches?

- -Eliminate branches
  - Predication (more on later)
- Delayed branch slot
  - SPARC, MIPS
- Or predict? ☺

Surviving Branches: Prediction

- Predict Branches
  - And predict them well!
- Fetch, decode, etc. on the predicted path
  - Option 1: No execute until branch resovled
  - Option 2: Execute anyway (speculation)
- Recover from mispredictions
  - Restart fetch from correct path

Branch Prediction

- Need to know two things
  - Whether the branch is taken or not (direction)
  - The target address if it is taken (target)
- Direct jumps, Function calls: unconditional branches
  - Direction known (always taken), target easy to compute
- Conditional Branches (typically PC-relative)
  - Direction difficult to predict, target easy to compute
- Indirect jumps, function returns
  - Direction known (always taken), target difficult

Branch Prediction: Direction

- Needed for conditional branches
  - Most branches are of this type
- Many, many kinds of predictors for this
  - Static: fixed rule, or compiler annotation (e.g. br.bwh (branch whether hint. IA-64))
  - Dynamic: hardware prediction
- Dynamic prediction usually history-based
  - Example: predict direction is the same as the last time this branch was executed

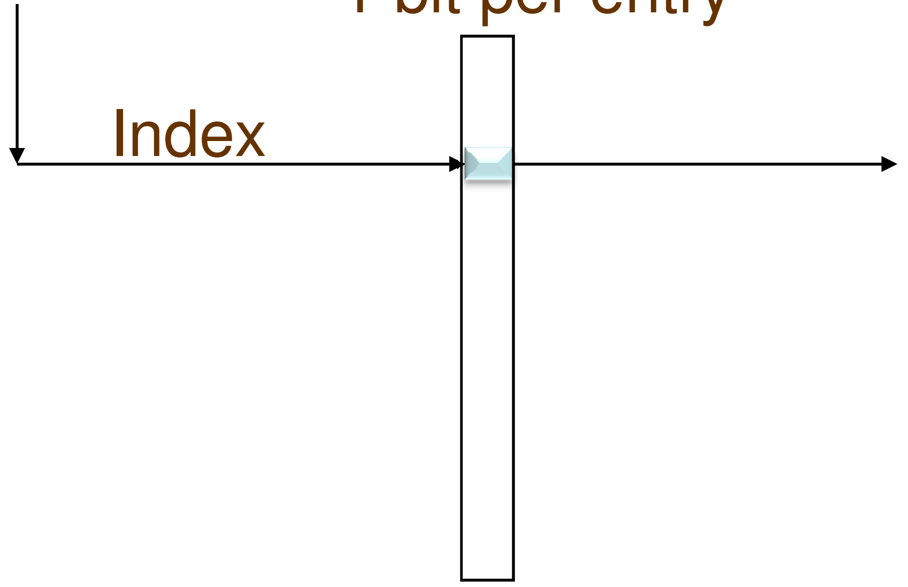
Static Prediction

- Always predict NT
  - easy to implement
  - 30-40% accuracy … not so good
- Always predict T
  - 60-70% accuracy
- BTFNT
  - loops usually have a few iterations, so this is like always predicting that the loop is taken
  - don't know target until decode

One-Bit Branch Predictor: Last-time predictor

K bits of branch instruction address

Branch history table of 2^K entries, 1 bit per entry



Use this entry to predict this branch:

0: predict not taken
1: predict taken

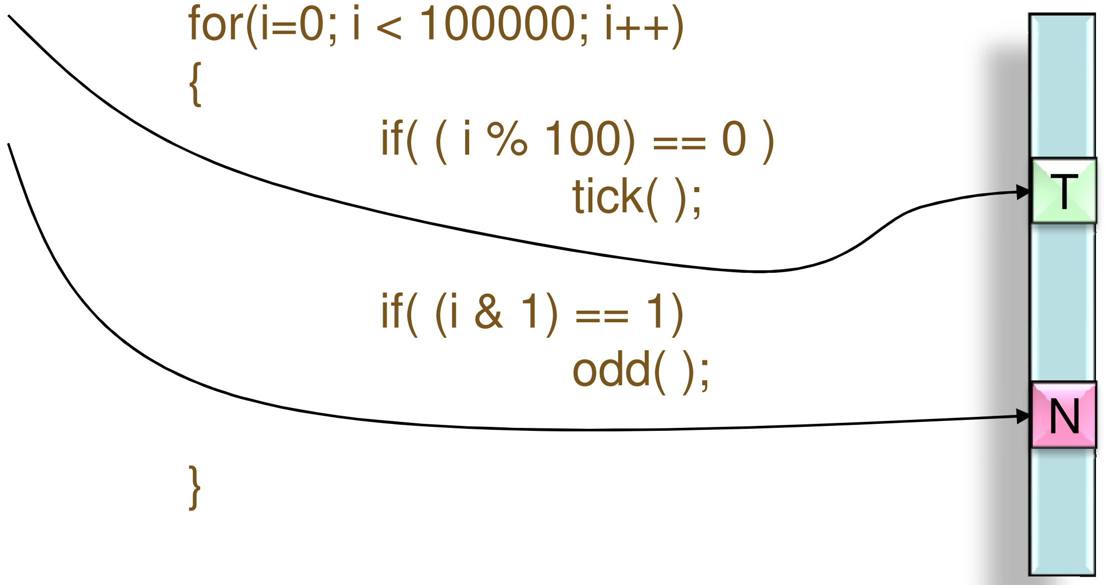
When branch direction resolved, go back into the table and update entry: 0 if not taken, 1 if taken

One-Bit Branch Predictor (cont'd)

```
0xDC08:          for(i=0; i < 100000; i++)
                 {
0xDC44:                  if( ( i % 100) == 0 )
                                 tick( );

0xDC50:                  if( (i & 1) == 1)
                                 odd( );

                 }
```



The Bit Is Not Enough!

- Example: short loop (8 iterations)
  - Taken 7 times, then not taken once
  - Not-taken mispredicted (was taken previously)

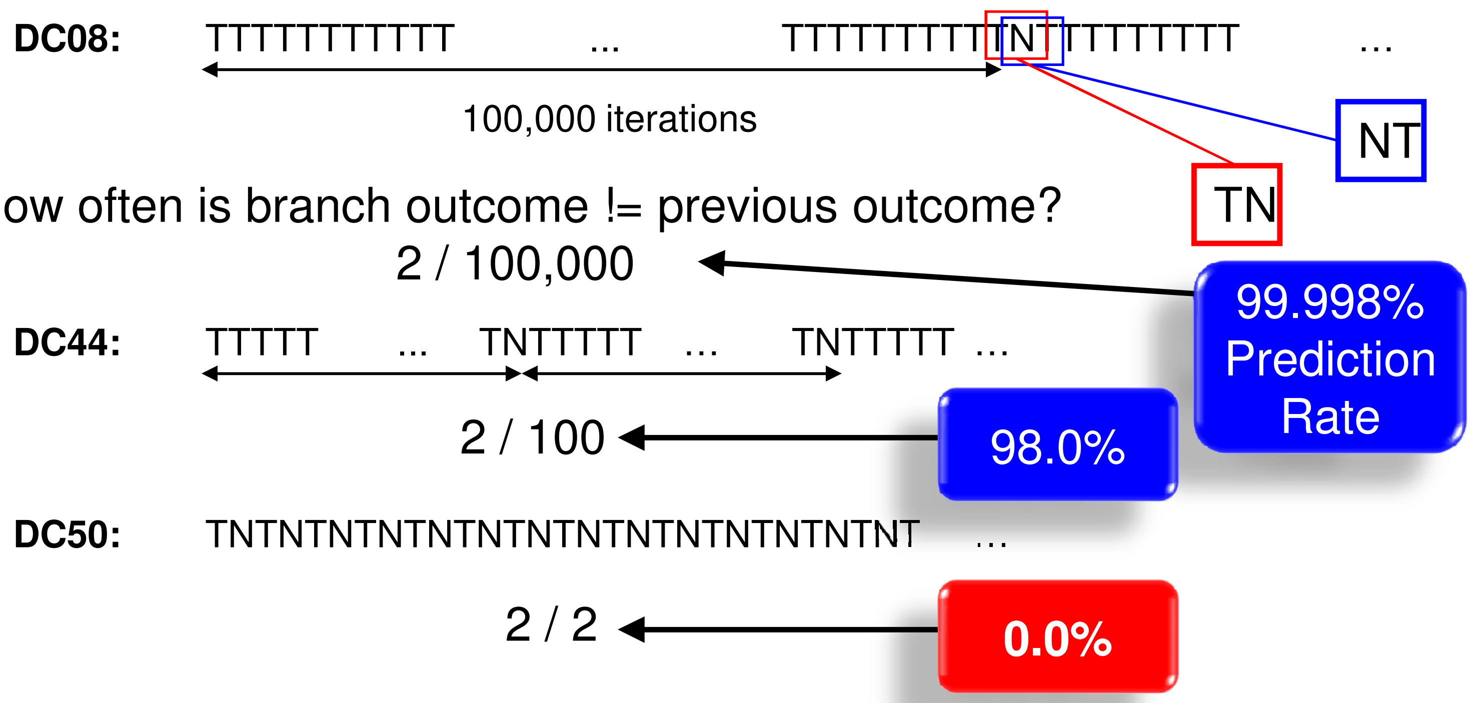
  Act: TTTTTTTNTTTTTTTNTTTTTTTTNT…

  Pred: XTTTTTTTNTTTTTTTNTTTTTTTN

  Corr: Xooooo MMooooooMMooooooMM

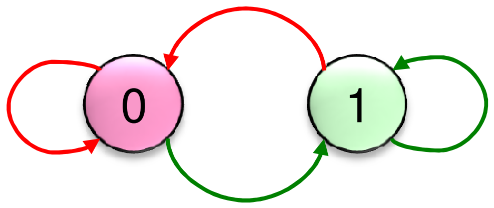
  Misprediction rate: 2/8 = 25%
- Execute the same loop again
  - First always mispredicted (previous outcome was not taken)
  - Then 6 predicted correctly
  - Then last one mispredicted again
- Each fluke/anomaly in a stable pattern results in two mispredicts per loop

Examples

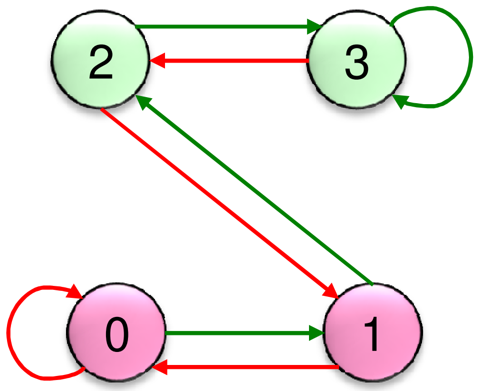

DC08:
TTTTTTTTTTTT ... TTTTTTTTTTTNTTTTTTTTT ...
100,000 iterations
NT
TN
How often is branch outcome != previous outcome?
2 / 100,000
99.998% Prediction Rate
DC44:
TTTTT ... TNTTTTT ... TNTTTTT ...
2 / 100
98.0%
DC50:
TNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNT ...
2 / 2
0.0%

Two Bits are Better Than One

●	Predict NT
●	Predict T
→	Transistion on T outcome
→	Transistion on NT outcome

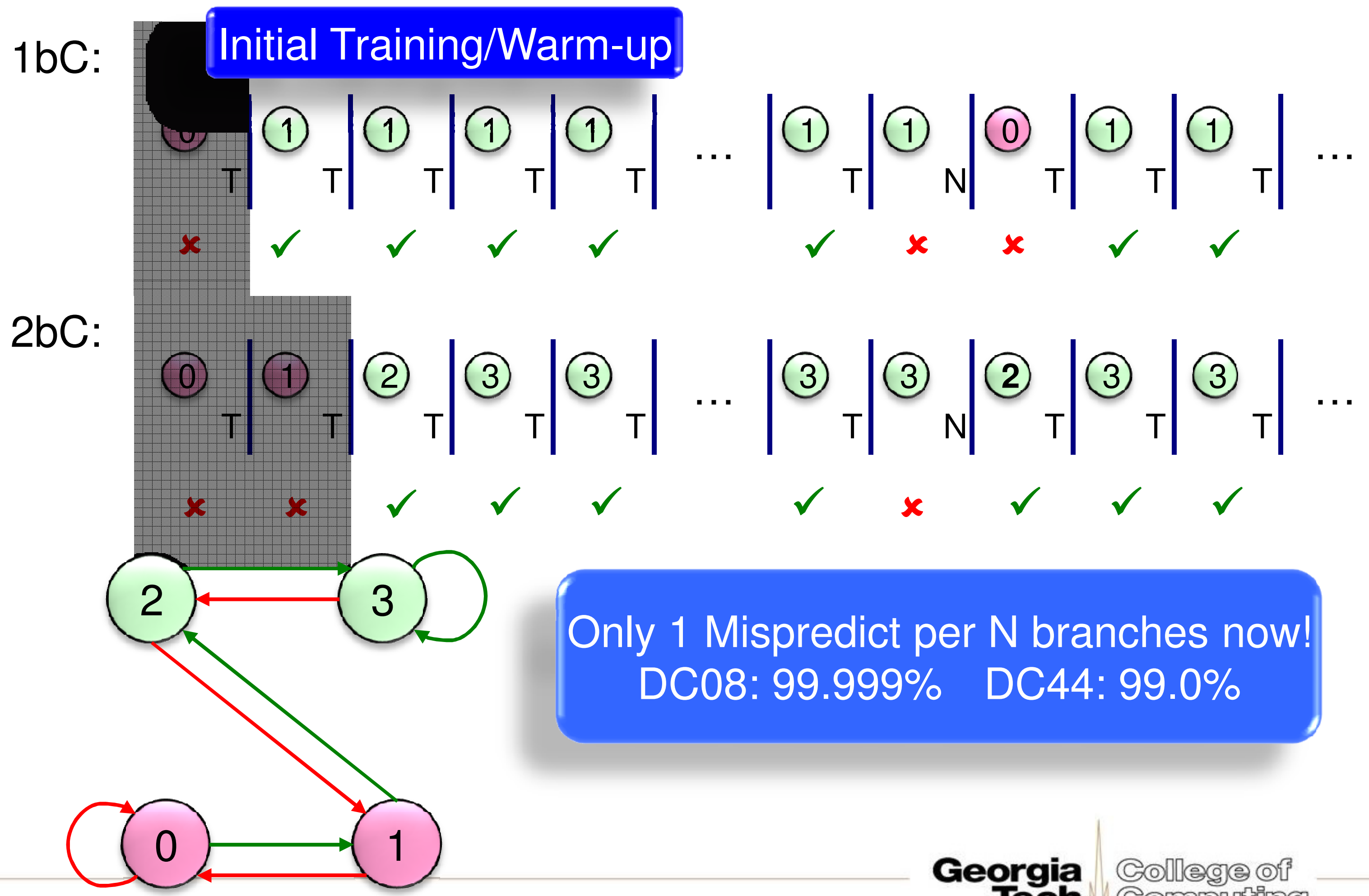


FSM for Last-time Prediction



FSM for 2bC
(**2**-**b**it **C**ounter)

Example



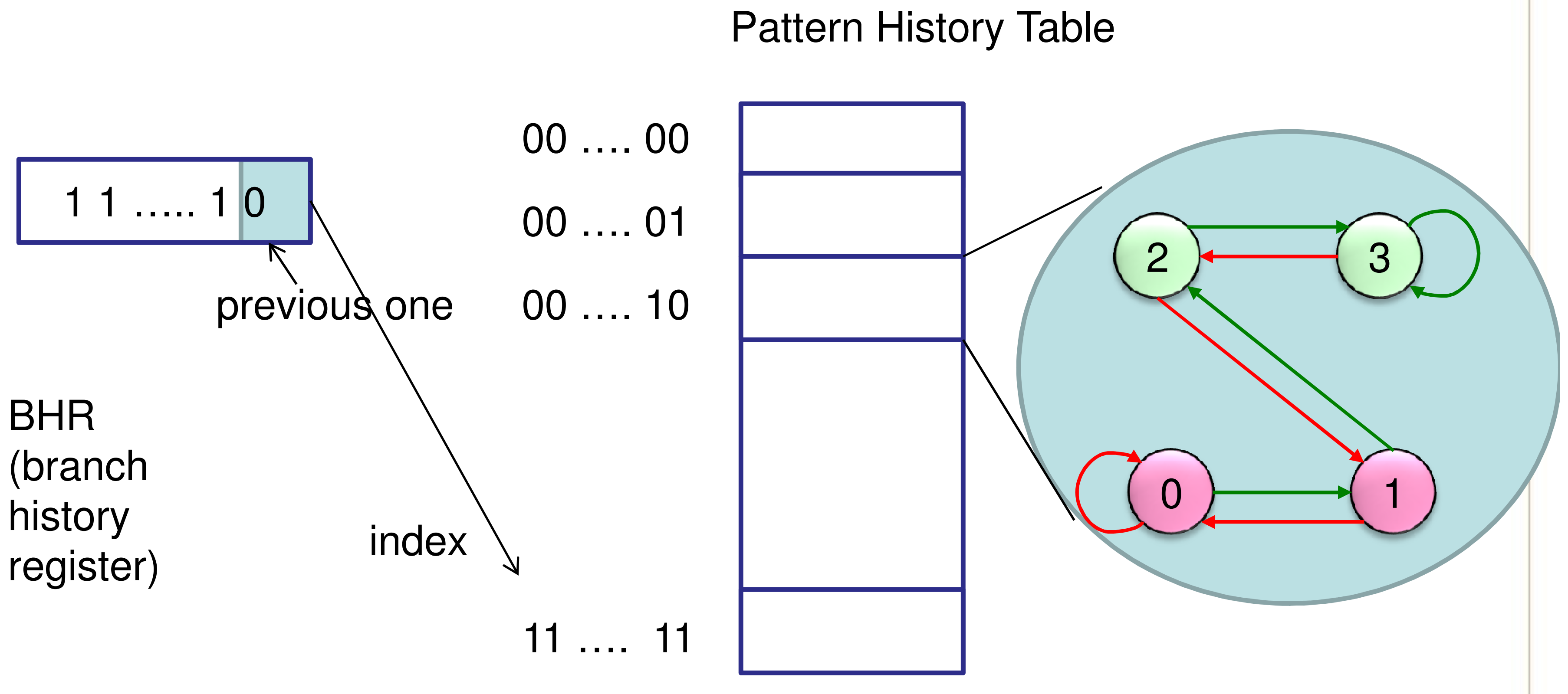
1bC:
Initial Training/Warm-up
1 T 1 T 1 T 1 T 1 T … 1 T 1 N 0 T 1 T 1 T …
✓ ✓ ✓ ✓ ✓ ✗ ✗ ✓ ✓
2bC:
0 T 1 T 2 T 3 T 3 T … 3 T 3 N 2 T 3 T 3 T …
✗ ✗ ✓ ✓ ✓ ✓ ✗ ✓ ✓ ✓
2
3
0
1

Only 1 Mispredict per N branches now!
DC08: 99.999%   DC44: 99.0%

Importance of Branches

- 98% → 99%
  - Who cares?
  - Actually, it's 2% misprediction rate → 1%
  - That's a halving of the number of mispredictions
- So what?
  - If a pipeline can fetch 5 instructions at a cycle and the branch resolution time is 20 cycles
  - To Fetch 500 instructions
  - 100 accuracy : 100 cycles
  - 98 accuracy:
    - 100 (correctly fetch) + 20 (misprediction)*10 = 300 cycles
  - 99 accuracy
    - 100 (correctly fetch) + 20 misprediction *5 = 200 cycles

Two-level Branch Predictor



Yeh&patt'92

Why does Global Predictor Work?

- Branches are correlated

Branch X: if (cond1)

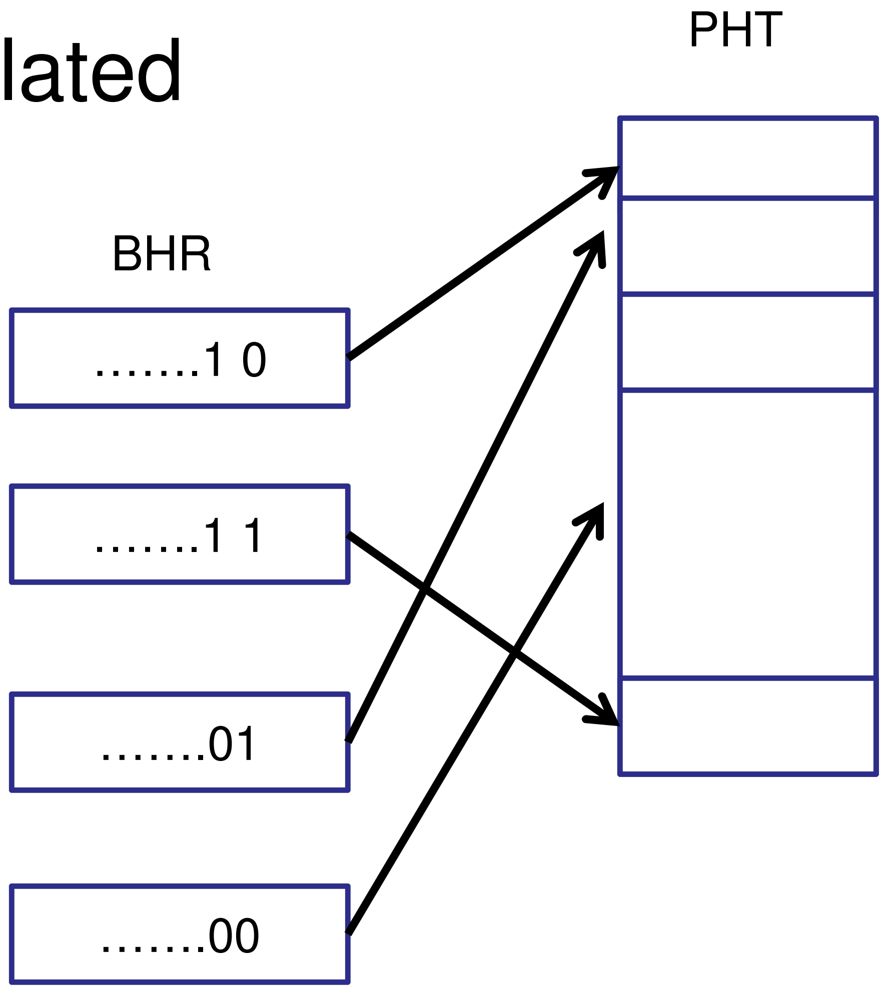
….

Branch Y: if (cond 2)

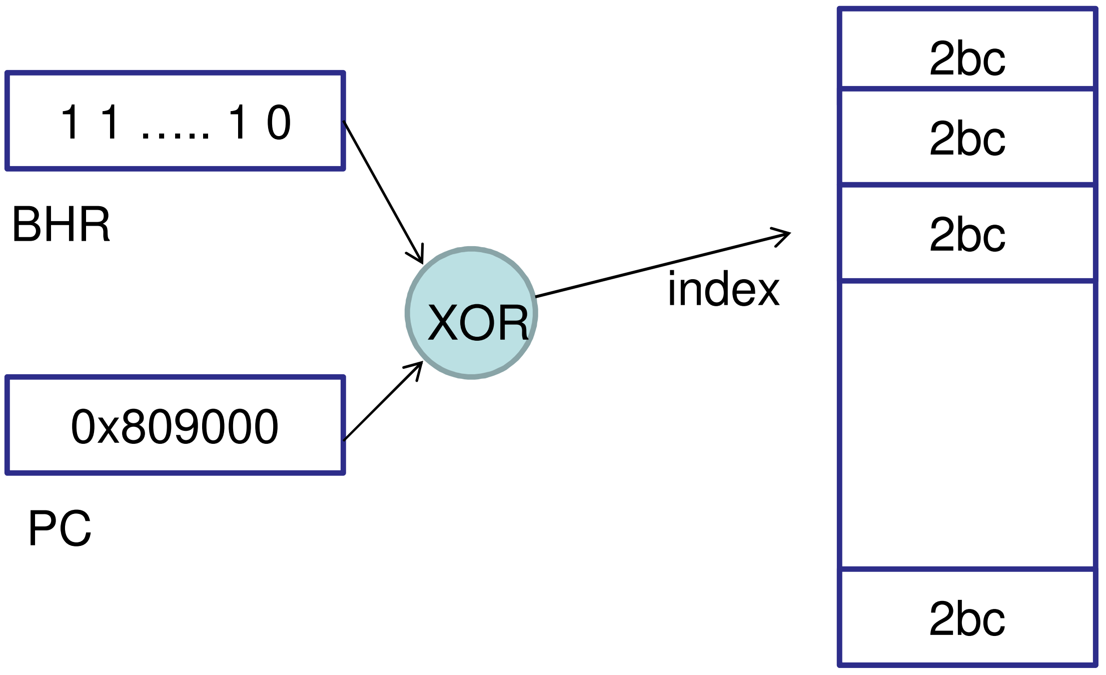
….

Branch Z : if (cond 1 and cond 2)

Branch X	Branch Y	Branch Z
1	0	0
1	1	1
0	1	0
0	0	0



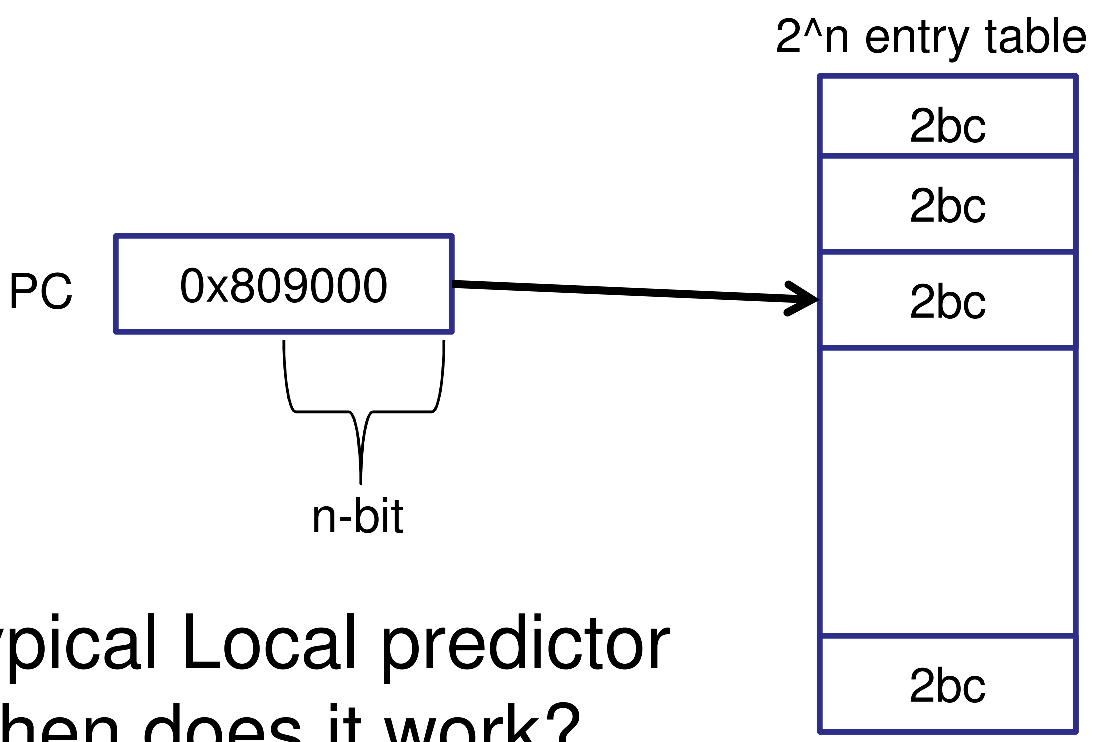
Gshare Branch Predictor



McFarling'93

Predictor size: 2^(history length)*2bit

Biomodal Branch Predictor



Typical Local predictor

When does it work?

-Loop,

-Repeat pattern

```
    a++;
   if (!(a%3)) { ..}
```

Target Address Prediction

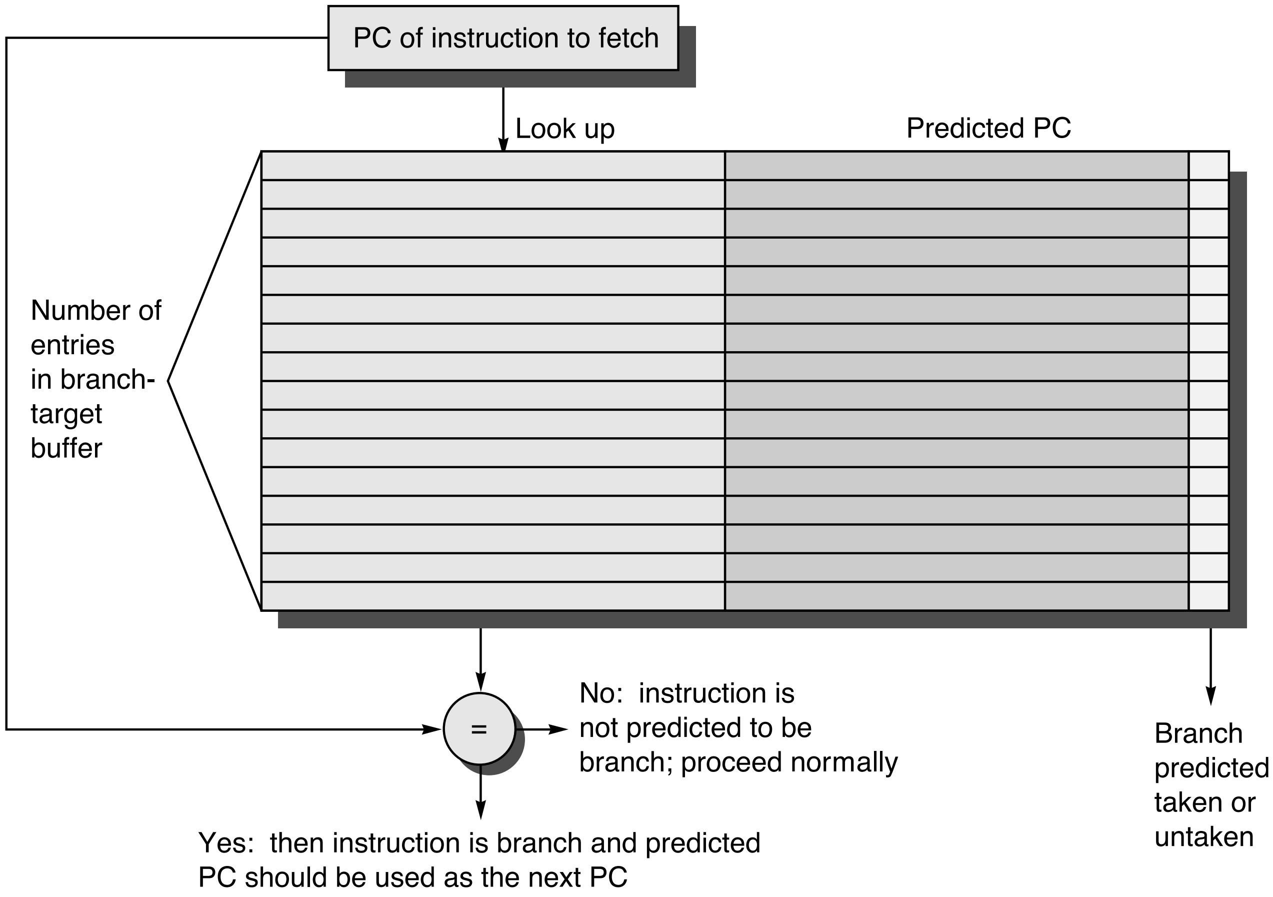
- Branch Target Buffer
  - IF stage: need to know fetch addr every cycle
  - Need target address one cycle after fetching a branch
  - For some branches (e.g., indirect) target known only after EX stage, which is way too late
  - Even easily-computed branch targets need to wait until instruction decoded and direction predicted in ID stage (still at least one cycle too late)
  - So, we have a fast predictor for the target that only needs the address of the branch instruction

Branch Target Buffer

- BTB indexed by instruction address (or fetch address)
- We don't even know if it is a branch!
- If address matches a BTB entry, it is *predicted to be a branch*
- BTB entry tells whether it is taken (direction) and where it goes if taken
- BTB takes only the instruction address, so while we fetch one instruction in the IF stage we are predicting where to fetch the next one from

Direction prediction can be factored out into separate table

Branch Target Buffer

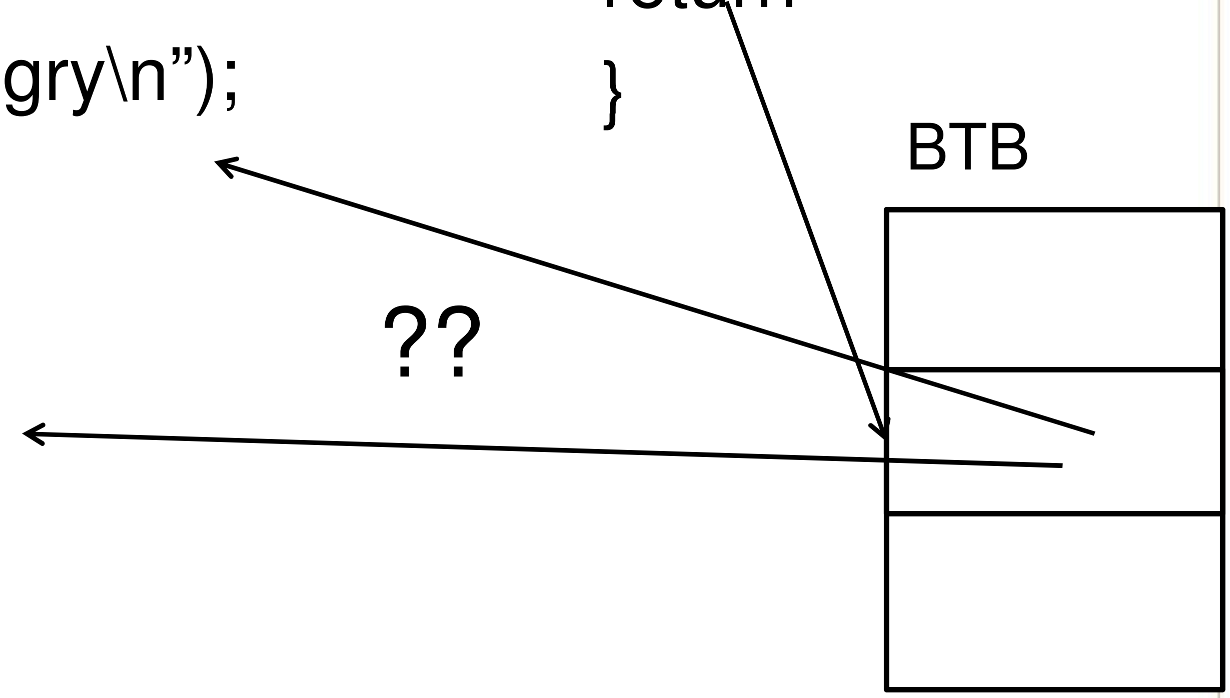


PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

Function Calls

```
main()
{
    foo();
    printf(“still hungry\n”);
    ….
    foo();
    printf(“full\n”);
}
```

```
foo(){
  …..
   return
   }
```



Return Address Stack (RAS)

- Function returns are frequent, yet
  - Address is difficult to compute (have to wait until EX stage done to know it)
  - Address difficult to predict with BTB (function can be called from multiple places)

Function Calls

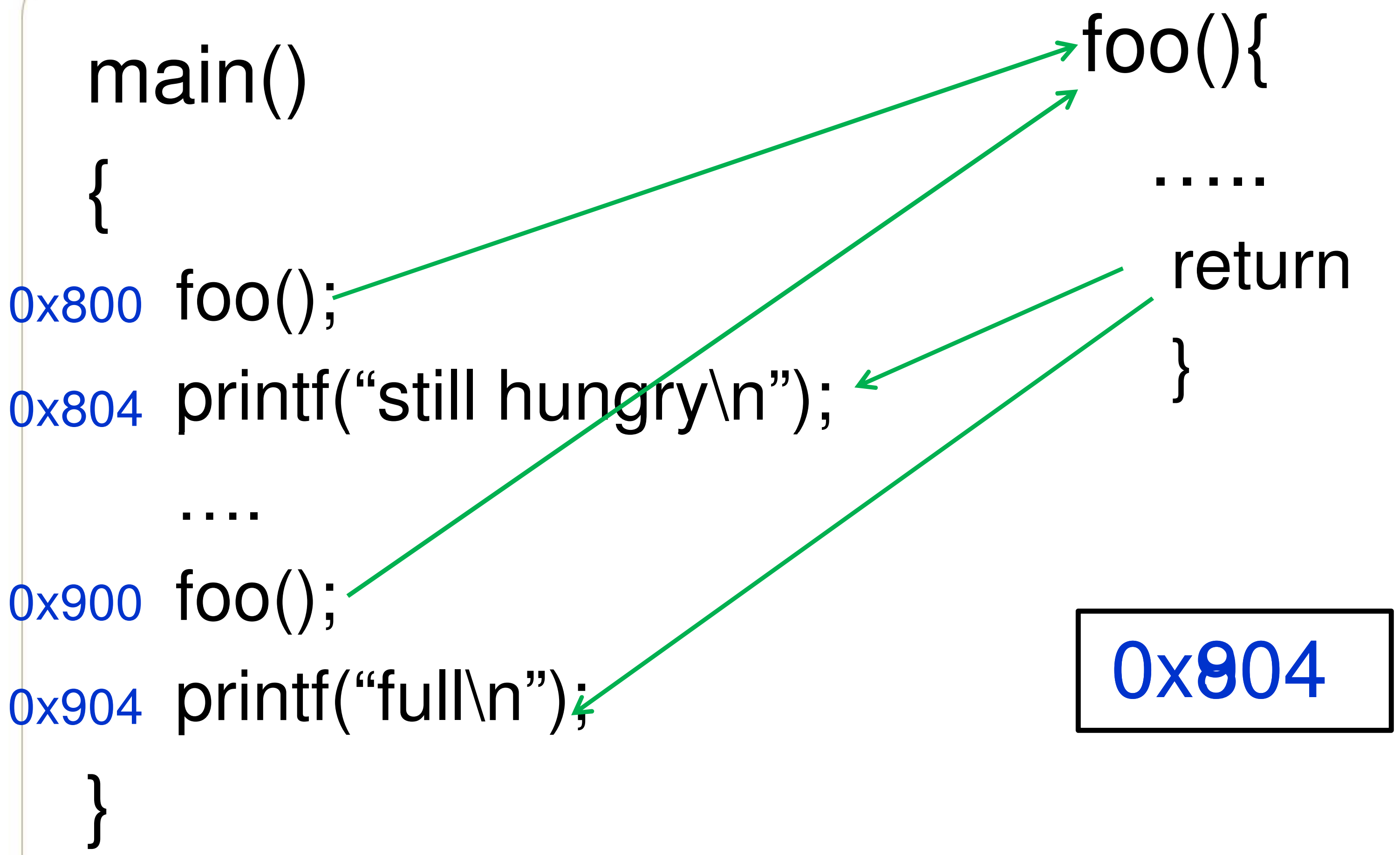
```
        main()
        {
0x800   foo();
0x804   printf(“still hungry\n”);
        ….
0x900   foo();
0x904   printf(“full\n”);
        }
```

```
foo(){
   …..
   return
   }
```

Return Address Stack (RAS)

- But return address is actually easy to predict
  - It is the address after the last call instruction that we haven't returned from yet
  - Hence the Return Address Stack

Function Calls

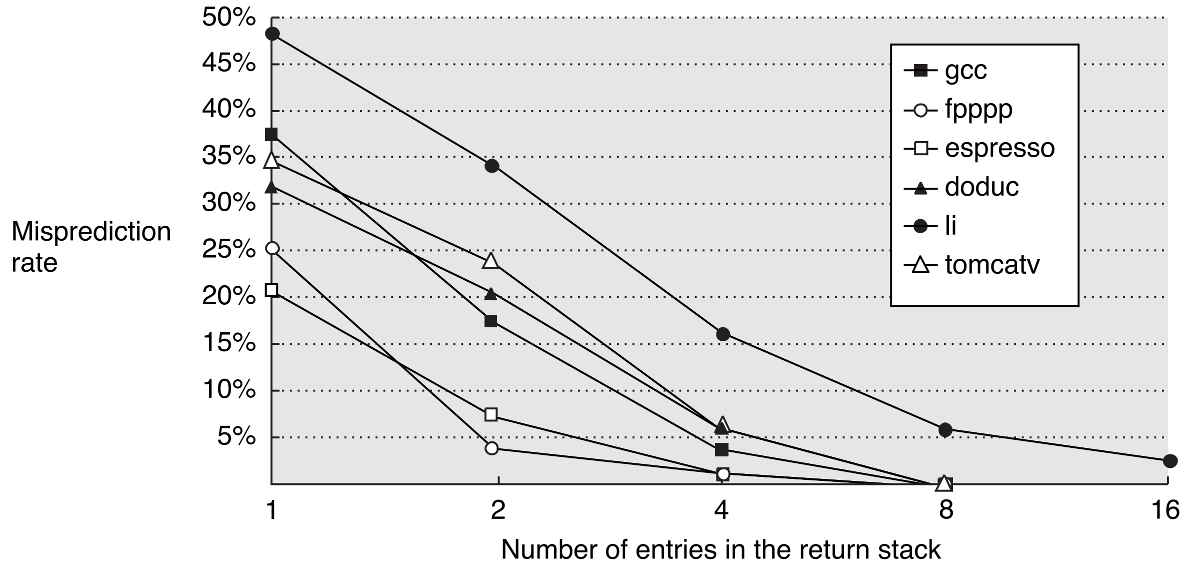


```
main()
{
0x800  foo();
0x804  printf("still hungry\n");
       ....
0x900  foo();
0x904  printf("full\n");
}
```

```
foo(){
   .....
   return
}
```

Return Address Stack (RAS)

- Call pushes return address into the RAS
- When a return instruction decoded,
  pop the predicted return address from RAS
- Accurate prediction even w/ small RAS



Georgia Tech College of Computing

LOOP Branches

```
for (ii =0; ii < 10; ii++)
{
 …
}
```

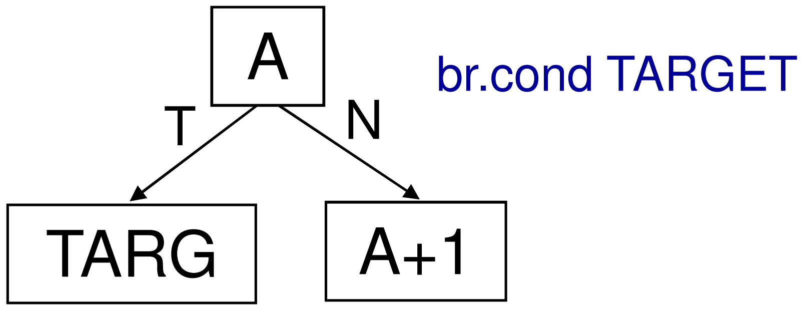
Loop branch is iterated 10 times all the time

- Special treatment for loop branches

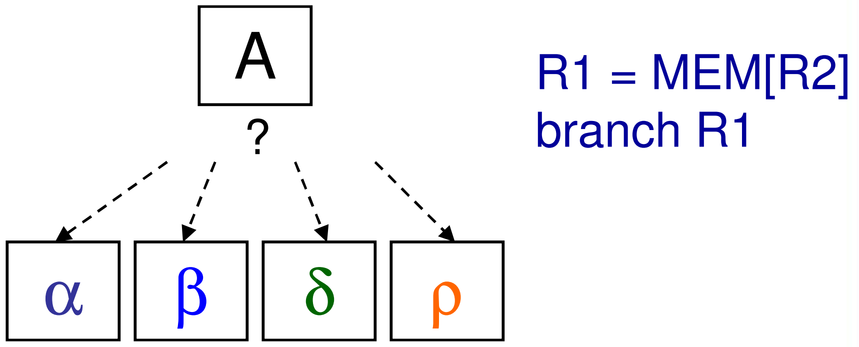
Options

- Prepare to branch (HPL-PD)
  - Software gives hints to the hardware about what the branch target will be. It saves us the target prediction since it has already been written into one of the target registers.
- Special Loop predictor (Intel's Pentium M)
  - Detect a loop branch
  - Train the max iteration counter value

Direct vs. Indirect Branch



Conditional (Direct) Branch



Indirect Branch

- Use the BTB
- A special indirect branch predictor (Intel's Core-2)

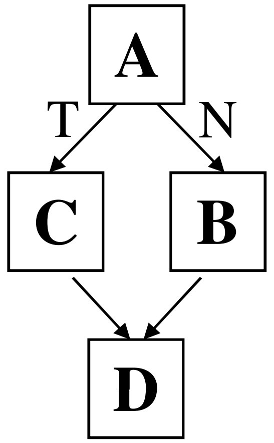
Eliminating Branches

- Predication
- Loop unrolling

Predication

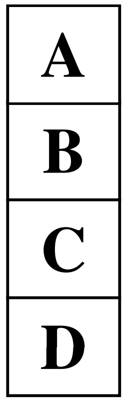
```
if (cond) {
      b = 0;
}
else {
      b = 1;
}
```

(normal branch code)



A
```
p1 = (cond)
branch p1, TARGET
```
B
```
mov b, 1
jmp JOIN
```
C
```
TARGET:
    mov b, 0
```

(predicated code)



A
```
p1 = (cond)
```
B
```
(!p1) mov  b, 1
```
C
```
(p1) mov  b, 0
```

Convert control flow dependency to data dependency

Pro: Eliminate hard-to-predict branches (in traditional architecture)

Eliminate branch divergence (in CUDA)

Cons: Extra instructions

Instruction Predication in G80

- Comparison instructions set condition codes (CC)
- Instructions can be predicated to write results only when CC meets criterion (CC != 0, CC >= 0, etc.)

- Compiler tries to predict if a branch condition is likely to produce many divergent warps
  - If guaranteed not to diverge: only predicates if < 4 instructions
  - If not guaranteed: only predicates if < 7 instructions
- May replace branches with instruction predication

- ALL predicated instructions take execution cycles
  - Those with false conditions don't write their output
    - Or invoke memory loads and stores
  - Saves branch instructions, so can be cheaper than serializing divergent paths

Loop Unrolling

- Transforms an M-iteration loop into a loop with M/N iterations
  - We say that the loop has been unrolled N times

```
for(i=0;i<100;i++)
  a[i]*=2;
```

$\Rightarrow$

```
for(i=0;i<100;i+=4){
  a[i]*=2;
  a[i+1]*=2;
  a[i+2]*=2;
  a[i+3]*=2;
}
```

Some compilers can do this (`gcc -funroll-loops`)
Or you can do it manually (above)

Why Loop Unrolling? (1)

- Less loop overhead

```
for(i=0;i<100;i++)
  a[i] += 2;
```

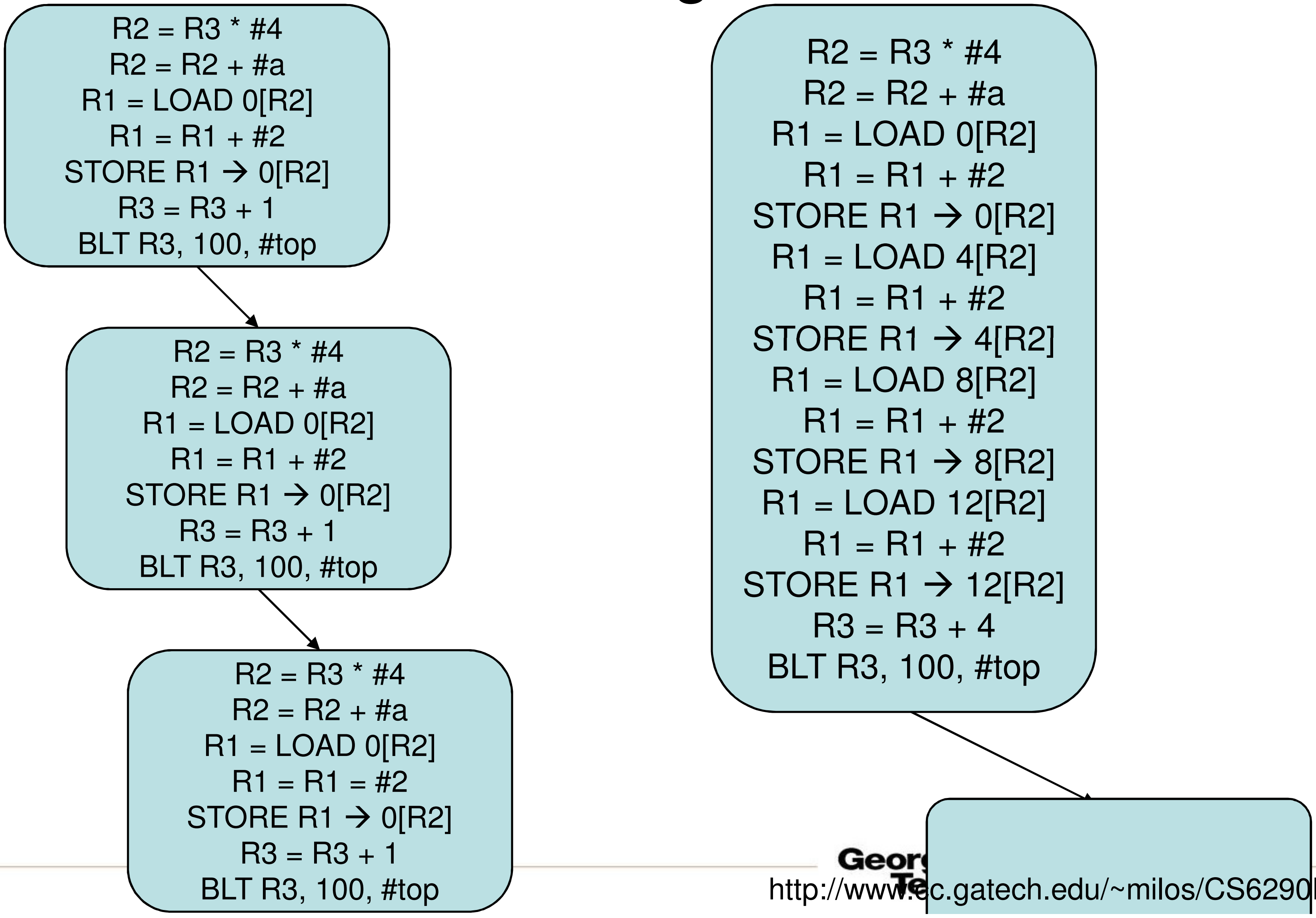
$\Rightarrow$

```
for(i=0;i<100;i+=4){
  a[i]   += 2;
  a[i+1] += 2;
  a[i+2] += 2;
  a[i+3] += 2;
}
```

How many branches?

Why Loop Unrolling? (2)

- Allows better scheduling of instructions



```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 = #2
STORE R1 → 0[R2]
R3 = R3 + 1
BLT R3, 100, #top
```

```
R2 = R3 * #4
R2 = R2 + #a
R1 = LOAD 0[R2]
R1 = R1 + #2
STORE R1 → 0[R2]
R1 = LOAD 4[R2]
R1 = R1 + #2
STORE R1 → 4[R2]
R1 = LOAD 8[R2]
R1 = R1 + #2
STORE R1 → 8[R2]
R1 = LOAD 12[R2]
R1 = R1 + #2
STORE R1 → 12[R2]
R3 = R3 + 4
BLT R3, 100, #top
```

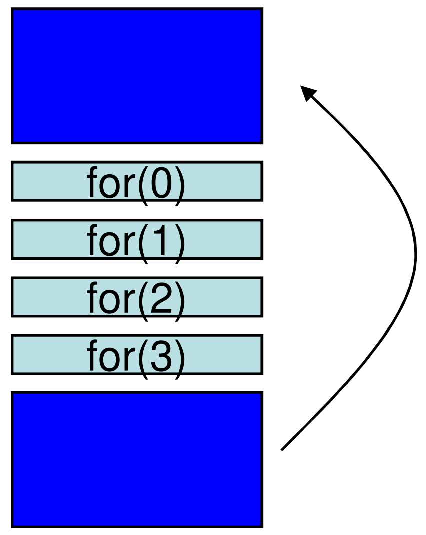
Why Loop Unrolling? (3)

- Get rid of small loops

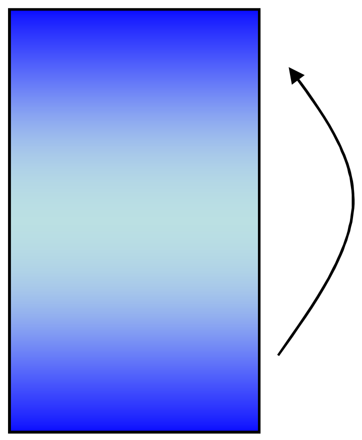
```
for(i=0;i<4;i++)
  a[i]*=2;
```

$\Rightarrow$

```
a[0]*=2;
a[1]*=2;
a[2]*=2;
a[3]*=2;
```



Difficult to schedule/hoist insts from bottom block to top block due to branches



Easier: no branches in the way

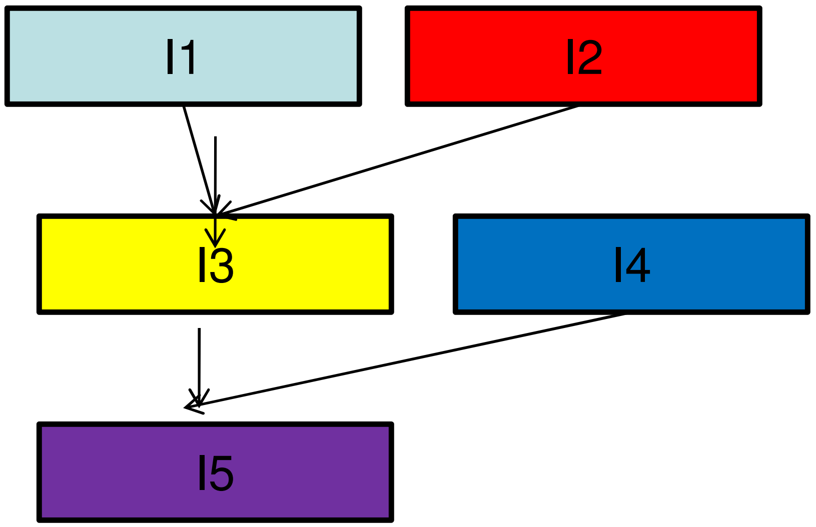
VLIW

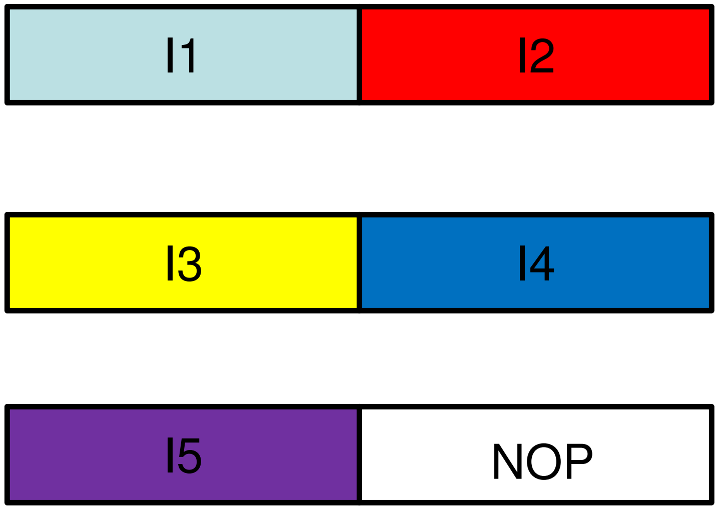
- VLIW = Very Long Instruction Word

Int Op 1	Int Op 2	Mem Op 1	Mem Op 2	FP Op 1	FP Op 2

- ***Everything*** is statically scheduled
  - All hardware resources exposed to compiler
  - Compiler must figure out what to do and when to do it
  - Get rid of complex scheduling hardware
  - More room for “useful” resources
- Examples:
  - Texas Instruments DSP processors
  - Transmeta’s processors
  - Intel IA-64 (EPIC), ATI Graphics processor

Static Instruction Scheduling





Why VLIW is good?

- Let the compiler do all of the hard work
  - Expose functional units, bypasses, latencies, etc.
  - Compiler can do its best to schedule code well
  - Compiler has plenty of time to do analysis
  - Compiler has larger scope (view of the program)
- Works extremely well on regular codes
  - Media Processing, Scientific, DSP, etc.
- Can be energy-efficient
  - Dynamic scheduling hardware is power-hungry

Why is VLIW hard?

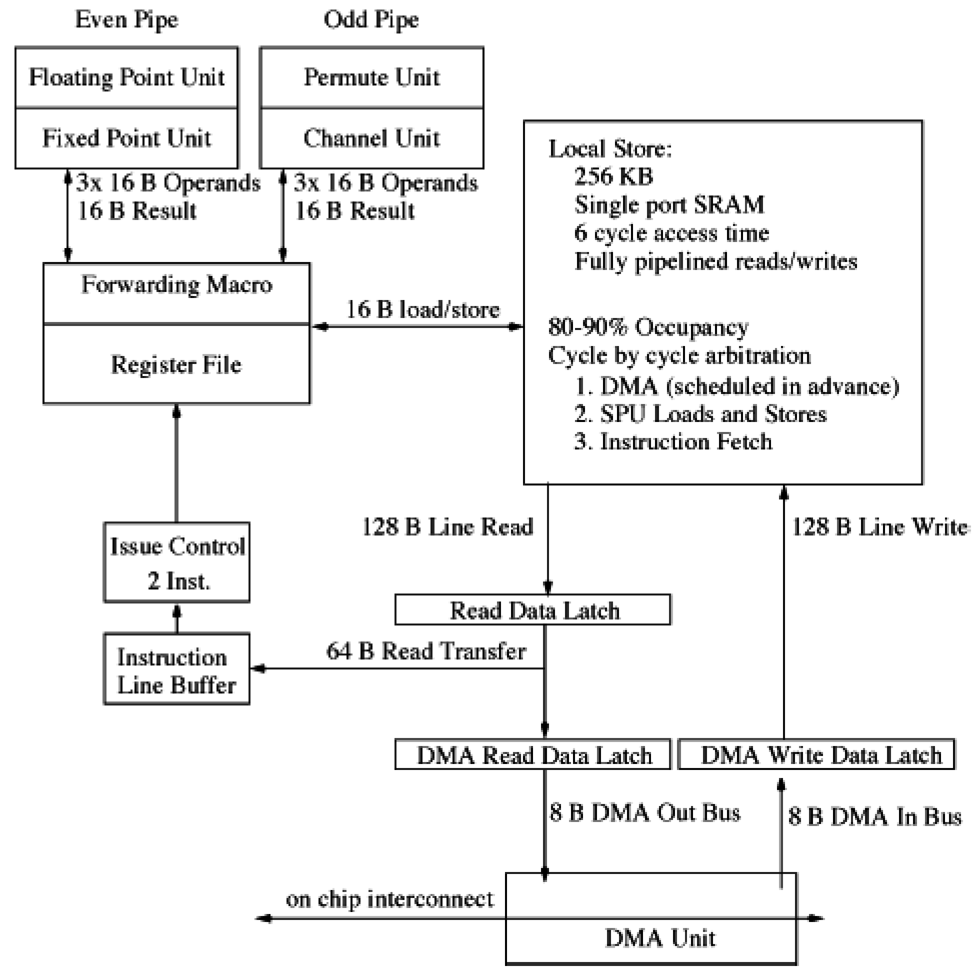
- Latencies are not constant
  - Statically scheduled assuming fixed latencies
- Irregular applications
  - Dynamic data structures (pointers)
  - "Common Case" changes when input changes
- Code can be very large
  - *Every resource exposed* also means that instructions are "verbose", with fields to tell each HW resource what to do
  - Many, many "NOP" fields
- 3wide VLIW machine → 6 wide VLIW machine?
- Where are instruction parallelism?

Extreme Example: Intel IA-64 (EPIC)

- Goal: Keep the best of VLIW, fix problems
  - Keep HW simple and let the compiler do its job
  - Support to deal with non-constant latencies
  - Make instructions more compact
- The reality
  - Compiler still very good at regular codes
  - HW among the most complex ever built by Intel
  - Good news: compiler still improving

SPE in Cell

- VLIWish


Even Pipe
Odd Pipe
Floating Point Unit
Fixed Point Unit
Permute Unit
Channel Unit
3x 16 B Operands
16 B Result
3x 16 B Operands
16 B Result
Forwarding Macro
Register File
16 B load/store
Local Store:
256 KB
Single port SRAM
6 cycle access time
Fully pipelined reads/writes
80-90% Occupancy
Cycle by cycle arbitration
1. DMA (scheduled in advance)
2. SPU Loads and Stores
3. Instruction Fetch
Issue Control
2 Inst.
Instruction Line Buffer
128 B Line Read
128 B Line Write
Read Data Latch
64 B Read Transfer
DMA Read Data Latch
DMA Write Data Latch
8 B DMA Out Bus
8 B DMA In Bus
on chip interconnect
DMA Unit