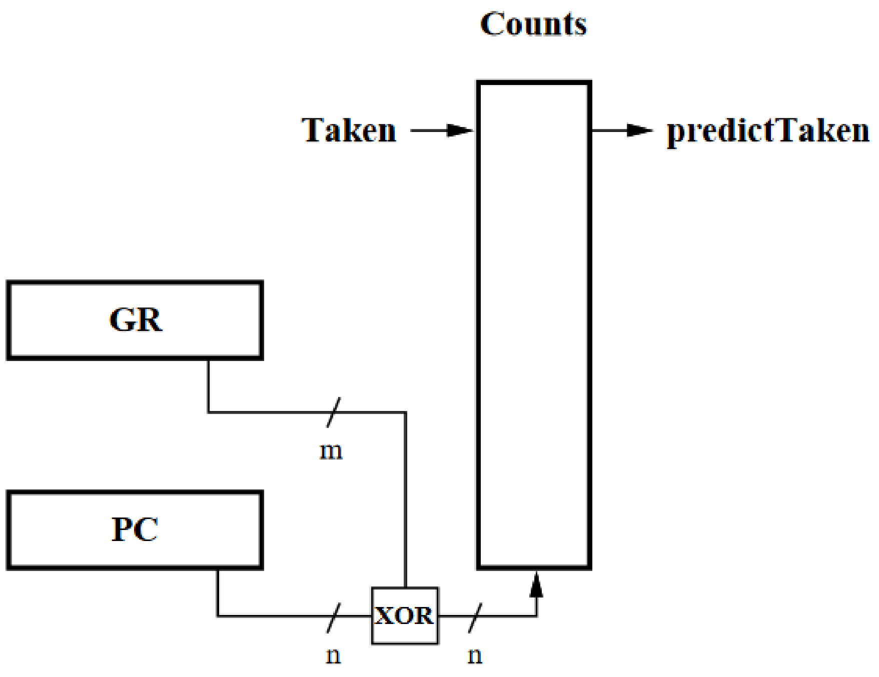
Branch Prediction Using Gshare

We have already known global history information weakly identifies the current branch. This suggests that there is a lot of redundancy in the counter index used by gselect. If there are enough address bits to identify the branch, we can expect the frequent global history combinations to be rather sparse. We can take advantage of this effect by hashing the branch address and the global history together. In particular, we can expect the exclusive OR of the branch address with the global history to have more information than either component alone. Moreover, since more address bits and global history bits are in use, there is reason to expect better predictions than gselect.
The following graph shows how gshare work:



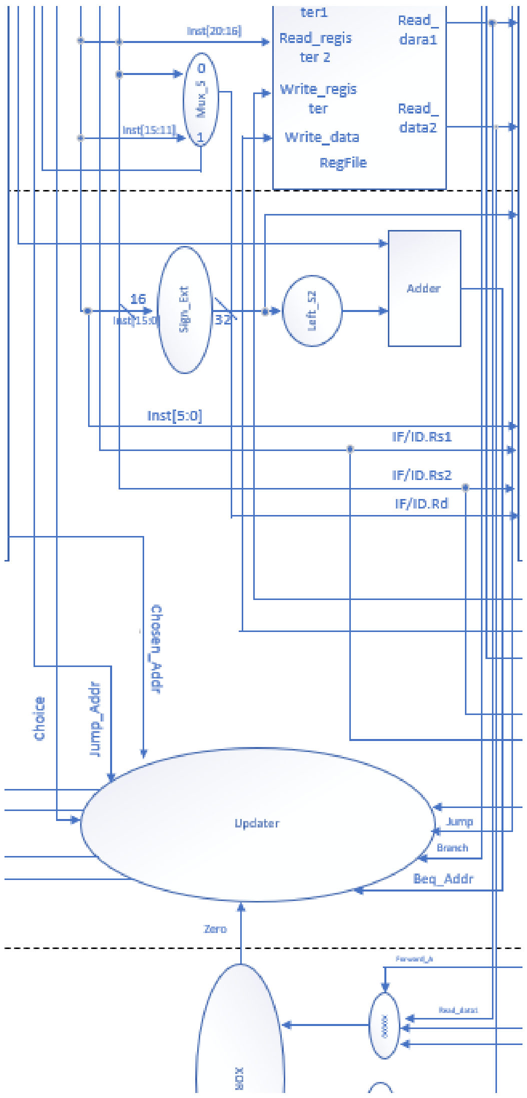
For example:The gshare predictor uses an 8 bit global branch history register (GHR). The most recent branch is stored in the least-significant-bit of the GHR and a value of '1' denotes a taken branch. The predictor XORs the GHR with bits [9:2] of the PC and uses this 8 bit value to index into a 256-entry pattern history table (PHT). Each entry of the PHT is a 2 bit saturating counter.
The gshare are used to combine the recent history and the current PC to make a prediction.if the indexed conter is 2'b11 or 2'b10 we predict this branch will be taken otherwise not taken.
How can we get the next pc for every clock?
At every fetch cycle, the predictor indexes into both the BTB and the PHT. If the predictor misses in the BTB (i.e., address tag != PC or valid bit = 0), then the next PC is predicted as PC+4. If the predictor hits in the BTB, then the next PC is predicted as the target supplied by the BTB entry when either of the following two conditions are met: (i) the BTB entry indicates that the branch is unconditional, or (ii) the gshare

predictor indicates that the branch should be taken. Otherwise, the next PC is predicted as PC+4.

What a BTB contains?

The branch target buffer (BTB) contains 1024 entries indexed by bits [11:2] of the PC. Each entry of the BTB contains (i) an address tag, indicating the full PC; (ii) a valid bit; (iii) a bit indicating whether this branch is unconditional; and (iv) the target of the branch.

To reduce latency for a branch resolving , I decide to reslove a branch in ID stage as flowing:



This is a part of design graph.

In this stage we calculate out the branch address and compare two intergers which come from RegFile or other stages(i.e. EX,MEM) by forwarding and decide wheather my previous pridiction is correct or not.

After resolving branch,we need to update. The update is consisting of: (i) updating the PHT, which is indexed using the current value of the GHR (ii) updating the GHR, and (iii) updating the BTB. Unconditional branches do not update the PHT or the GHR, but only the BTB (setting the unconditional bit in the corresponding entry.

What if we have a misprediction?
The correct predictions can be viewed as normal instructions(not branch) and continue to execute the following instructions without any delay!
Unfortunately,if we have encounterd a misprediction we must do some remedy.First flush the pipeline,second fetch the instruction from the correct address.

When to flush the pipeline?
When resolving a branch, the pipeline is flushed under any of the following conditions:

- The instruction is a branch, but the predicted direction does not match the actual direction.
- The instruction is a branch, and it is taken, but the predicted destination (target) does not match the actual destination
- The instruction is a branch, but it was not recognized as a branch (i.e., BTB miss)

One more thing: All branch predictor structures are initialized to 0.

We have already discussed the concepts of a gshare branch predictor. Let's talk about some details:
How do I remedy if there is a misprediction?
IF_ID_Flush is a signal for this usage.if the true direction doesn't match the predict direction we set this signal to 1. This signal can flush the IF/ID pipeline register and indicate the BTB to choose the check_Addr as next PC.
How do I use forward and stall mechanism to solve data hazard when a beq instruction is preceded by a R-type or I-type instruction (RAW)?
Consider the following example:

```
1:Lw R2 2(R1)
  Beq R2 R0 -4

2:Add R2 R1 R1
   Beq R2 R0 -4
```

The first example illustrate a RAW dependence between Beq and Lw when access to R2 register.Since we decide to resolve a branch at ID stage and Lw happens at Mem stage so we need to stall two clock cycles!

The second example illustrate a RAW dependence between Beq and Add when access to R2 register.Since we decide to resolve a branch at ID stage and Add happens at EX stage so we need to stall only one clock cycle!

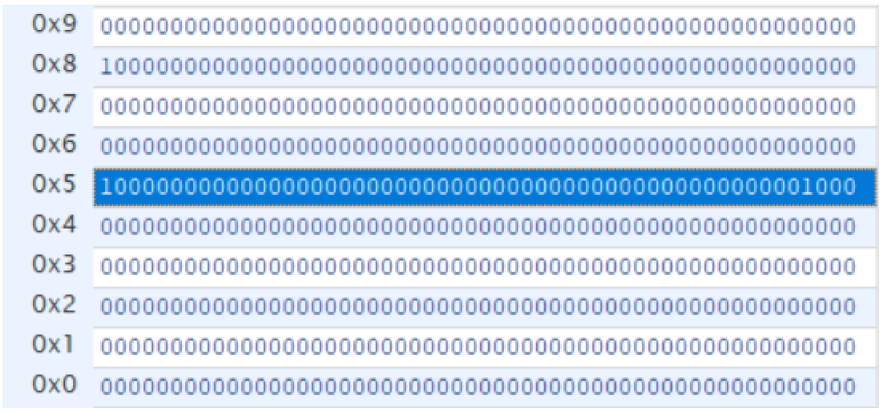
These function can be implemented by Hazard detection Unit and Forwarding Unit which are also used for data hazard between two non-branch instructions.

Let’s see the performance of gshare branch predictor now:

```
`ifdef BRANCH_TEST    //R5 is i ,R1 is j
Rom[255]<=32'hffff_ffff;//idle instruction
Rom[0]<=32'b00000000101000100010100000100000;    //Add R5 R5 R2
Rom[1]<=32'b00000000001000010000100000100010; // Sub R1 R1 R1=>R1=0
Rom[2]<=32'b00000000001000100000100000100000;    //Add R1 R1 R2
Rom[3]<=32'b00000000010000001000110000101010;        //        slt        R3        R4
R1=>R3=R4<=R1?1:0
Rom[4]<=32'hffff_ffff;
Rom[5]<=32'b00010000000000011111111111111101; // beq R0 R3 -3
Rom[6]<=32'b00000000110001010011100000101010;        //        slt        R7        R6
R5=>R7=R6<=R5?1:0
Rom[7]<=32'hffff_ffff;
Rom[8]<=32'b00010000000000011111111111111000; // beq R0 R7 -8
`endif
```

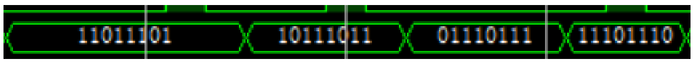
These 9 instructions(contains two idle instructions to avoid data hazard since we now focus on pridiction correct rate ,eliminating redundant data hazard will save us from stalls) are translated from:

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



0x9	0000000000000000000000000000000000000000000000000000000000000000
0x8	1000000000000000000000000000000000000000000000000000000000000000
0x7	0000000000000000000000000000000000000000000000000000000000000000
0x6	0000000000000000000000000000000000000000000000000000000000000000
0x5	1000000000000000000000000000000000000000000000000000000000001000
0x4	0000000000000000000000000000000000000000000000000000000000000000
0x3	0000000000000000000000000000000000000000000000000000000000000000
0x2	0000000000000000000000000000000000000000000000000000000000000000
0x1	0000000000000000000000000000000000000000000000000000000000000000
0x0	0000000000000000000000000000000000000000000000000000000000000000

The BTB shows 0x5 and 0x8 are valid and its target address are 8 and 0 respectively.



GHR has only 4 different choice:11011101 10111011 01110111 11101110

If we xor 00000101 with 11011101,10111011,01110111 we get 0xD8,0xBE,0x72
If we xor 00001000 with 11101110 we get 0xE6

Let’s test if the PHT matches the table below:

test	value	GR	result
j<3	j=1	1101	taken
j<3	j=2	1011	taken
j<3	j=3	0111	not taken
i<100		1110	usually taken

0xE6 11 , 0xD8 11 , 0xBE 11 , 0x72 00 .

Wow ,after we iterate several times ,we get this expecting PHT value.
This prediction correct rate is very high! Actually,excepet for only a few (i.e. 12)earlier mispredictions the later are all correct! If we have much **outer** iteration times we can get a high correct rate.

Here,it seems that we have finished our discussion about gshare branch predictor.But our test example are not convinceive enough,let’s think about the following questions:
First:
What if the inner iteration times are lager than the outer iteration times(inner iteration times are still smaller than GHR bits-1 which is 7)?
In this case we still can capture all patterns but the outer iteration times are limited,so miss rate is relatively higher(Remember the earlier several iterations we tend to predict incorrectly.)

What if the inner iteration times are lager than GHR bit -1(which is 7) ?
In this case we can’t capture all patterns.But this is not a problem,since many different pattern will have the same result: Taken.
And miss rate will become smaller and smaller if we continue increase **inner** iteration times.(Pay attention please,it’s inner iteration times rather than outer)

Now, we have really finshed our discussion about gshare predictor.