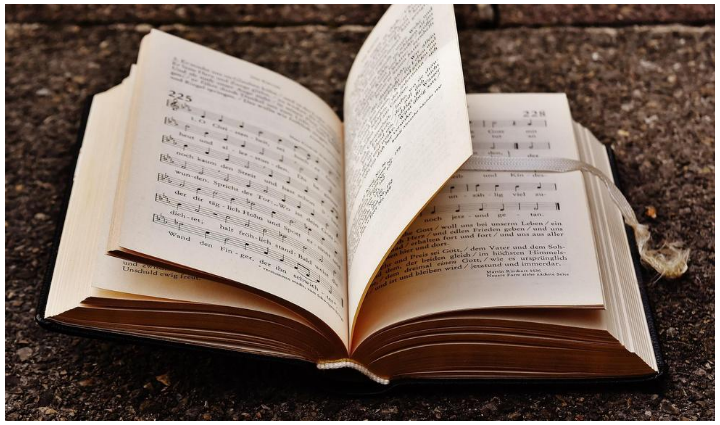
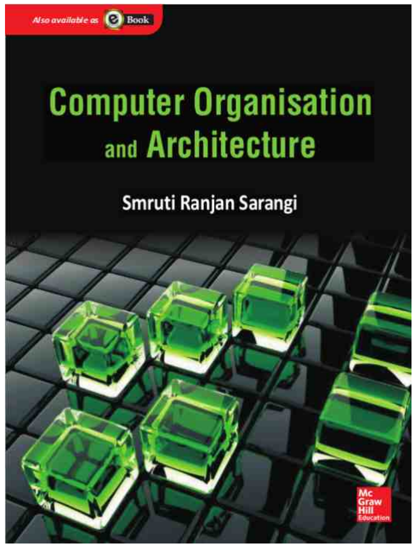


Chapter 3:
e Fetch and Decode Stages

Background Required to Understand this Chapter

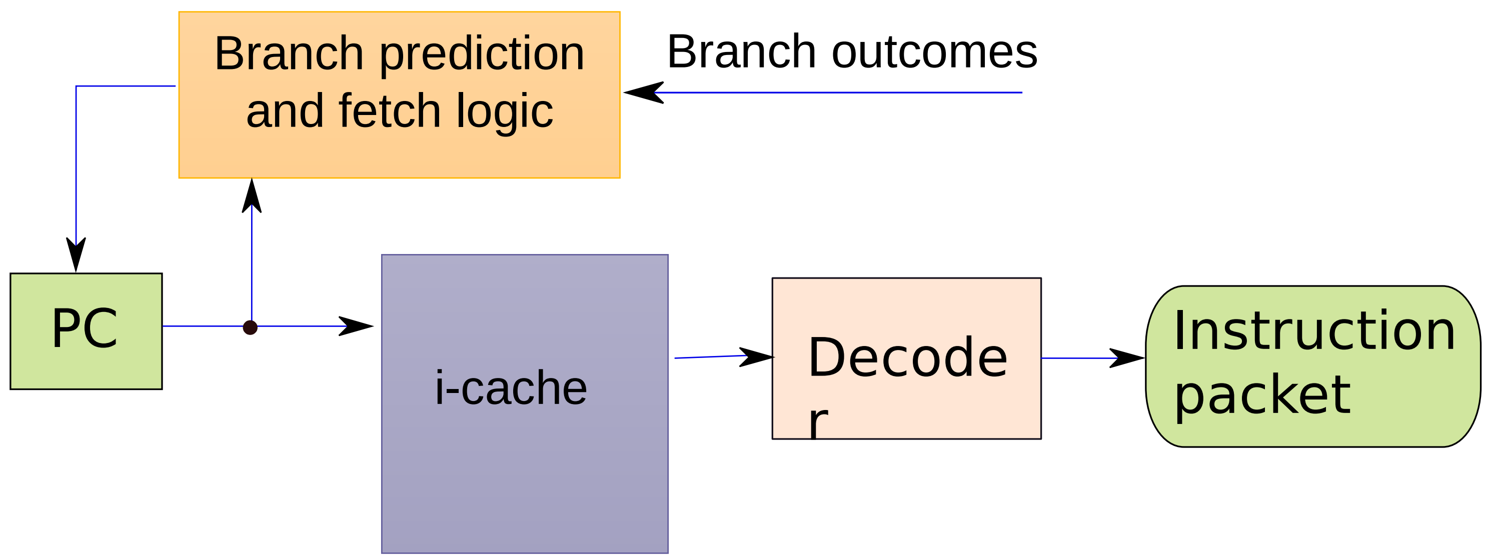
- Basic Boolean Logic
- Need for Branch Prediction
- RISC and CISC ISAs



http://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

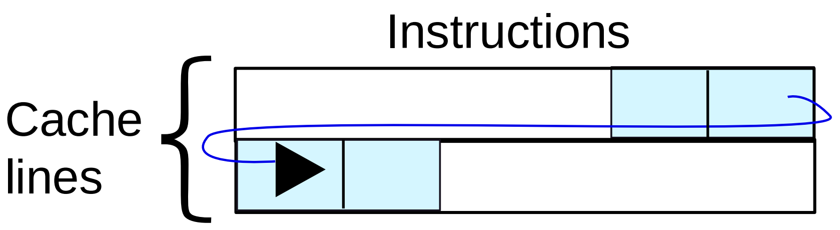
Background

The Fetch Stage

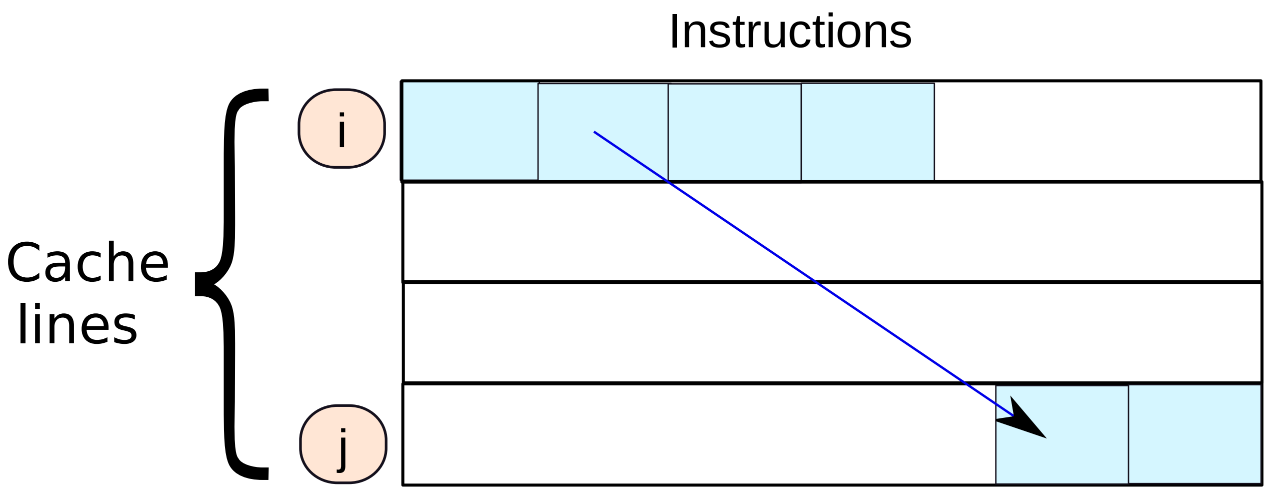


- The branch prediction and fetch logic computes the addresses of the instructions to be fetched in the next cycle
- The corresponding instructions are fetched from the i-cache
- They are sent down the pipeline

Reading 4 instructions in one go from the i-cache

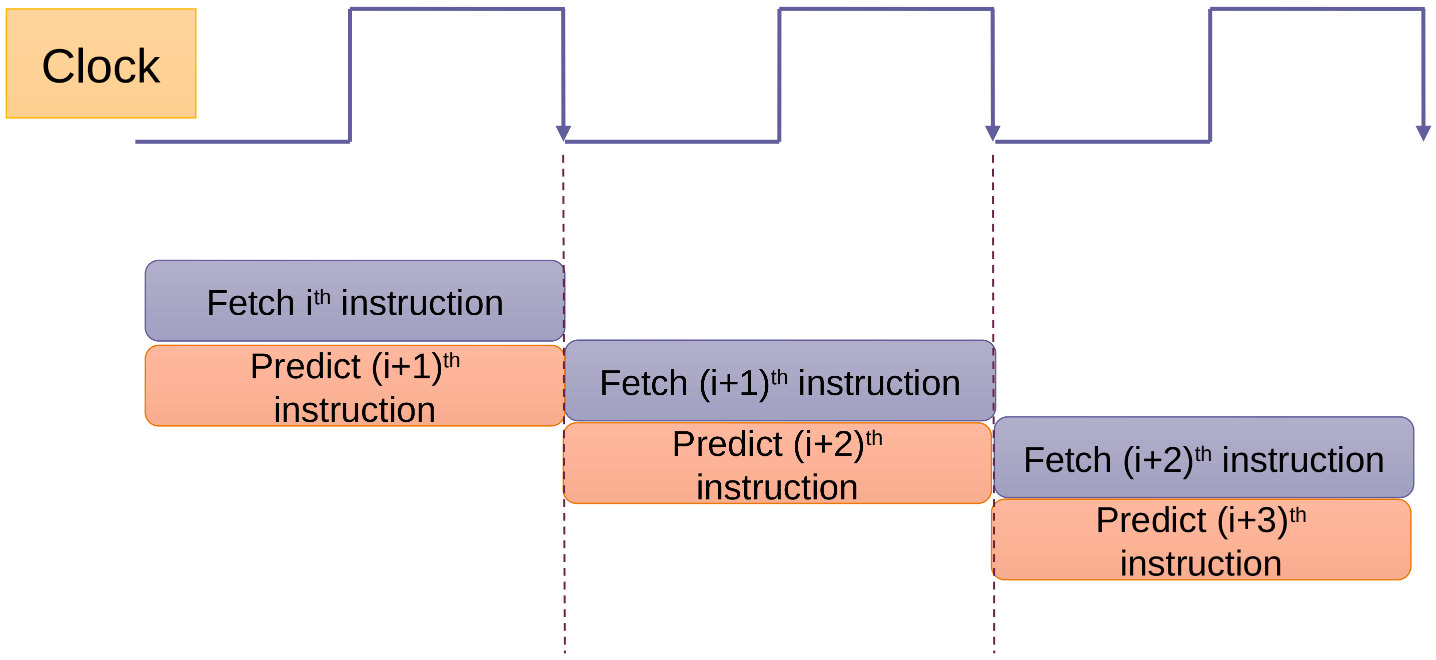


Assume no branches



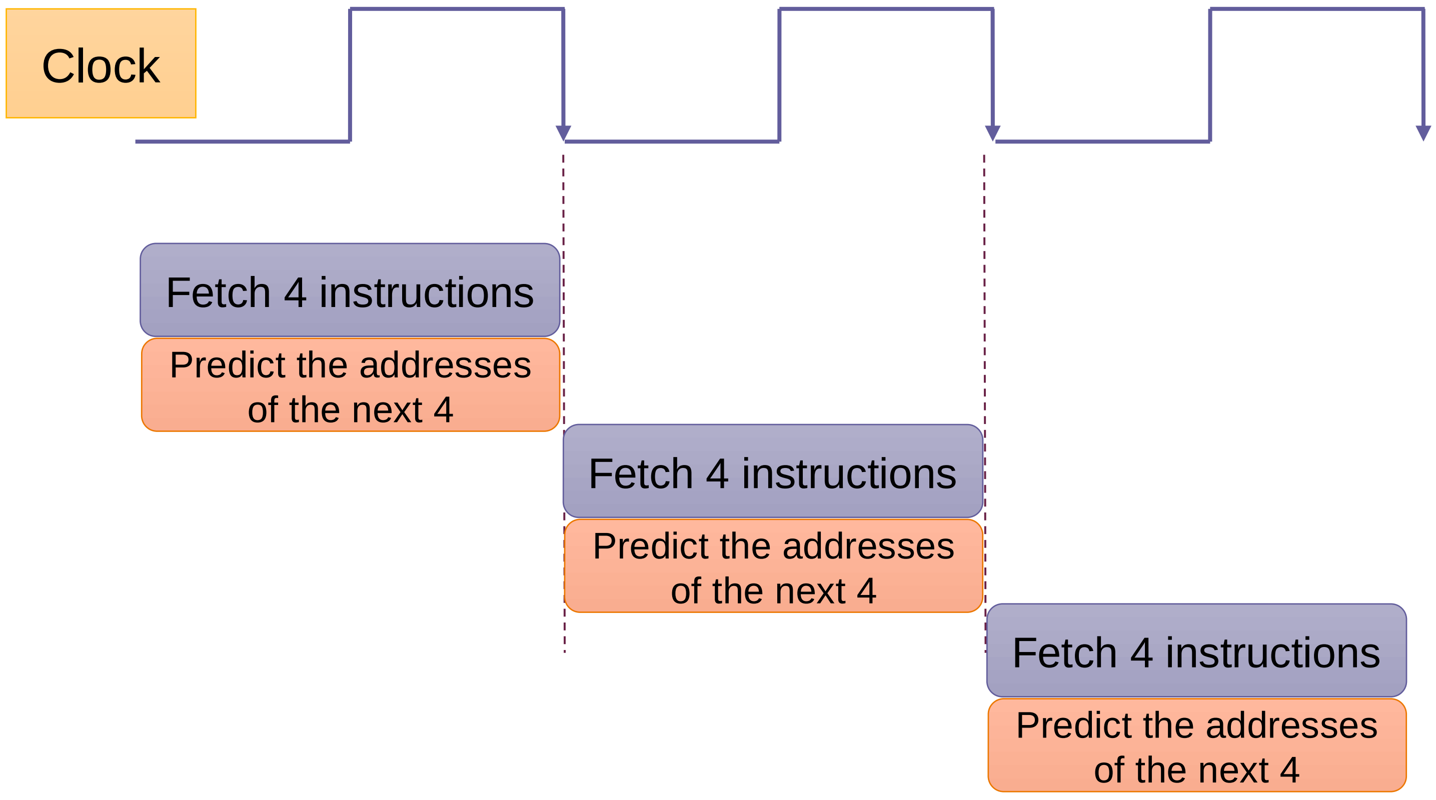
They might be in different cache lines

Timing: Fetching a single instruction per cycle


Clock
Fetch ith instruction
Predict (i+1)th instruction
Fetch (i+1)th instruction
Predict (i+2)th instruction
Fetch (i+2)th instruction
Predict (i+3)th instruction

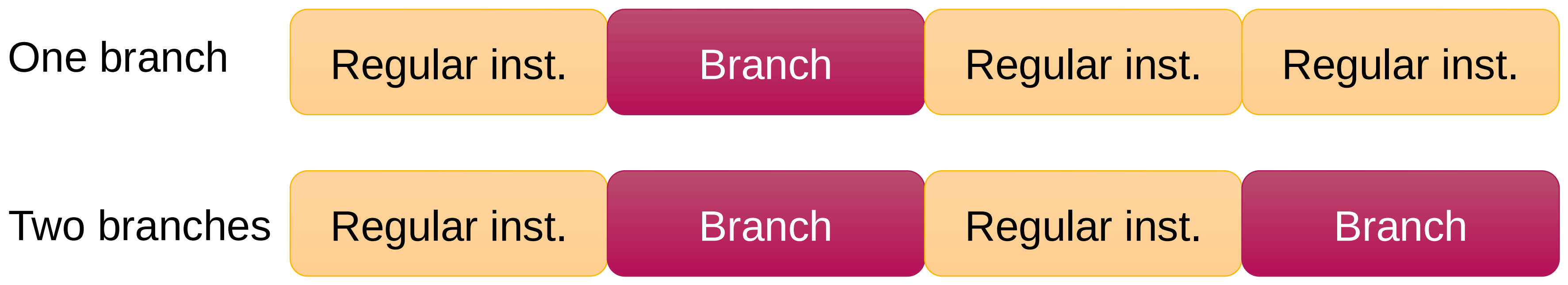
There is no time to look at an instruction. We just know its PC.

Timing: Fetching 4 instructions simultaneously


Clock
Fetch 4 instructions
Predict the addresses of the next 4
Fetch 4 instructions
Predict the addresses of the next 4
Fetch 4 instructions
Predict the addresses of the next 4

This is very hard to do. Why?

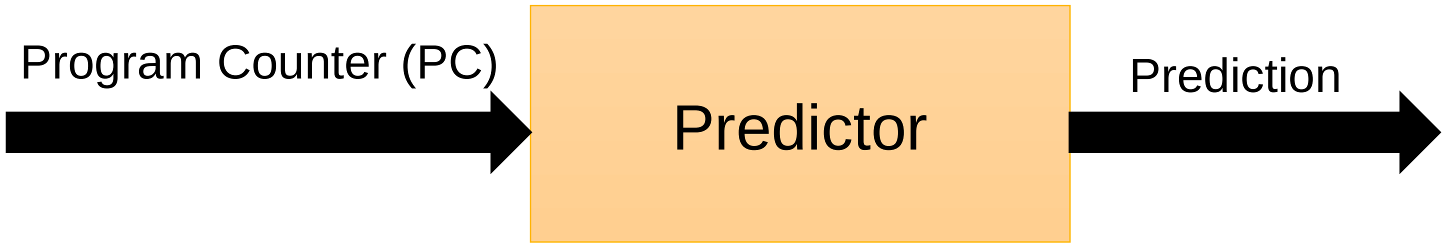
Branches in a Bundle of 4 Instructions



- If we have branches in a bundle of 4 instructions, the instructions will not be contiguous
- To predict the second branch, we need to first predict the first branch

Branch Prediction: Three Problems

- Predict if an instruction is a branch or not.
- If it is a branch, predict is direction.
- Predict its target.



Contents

Is an instruction a branch or not?

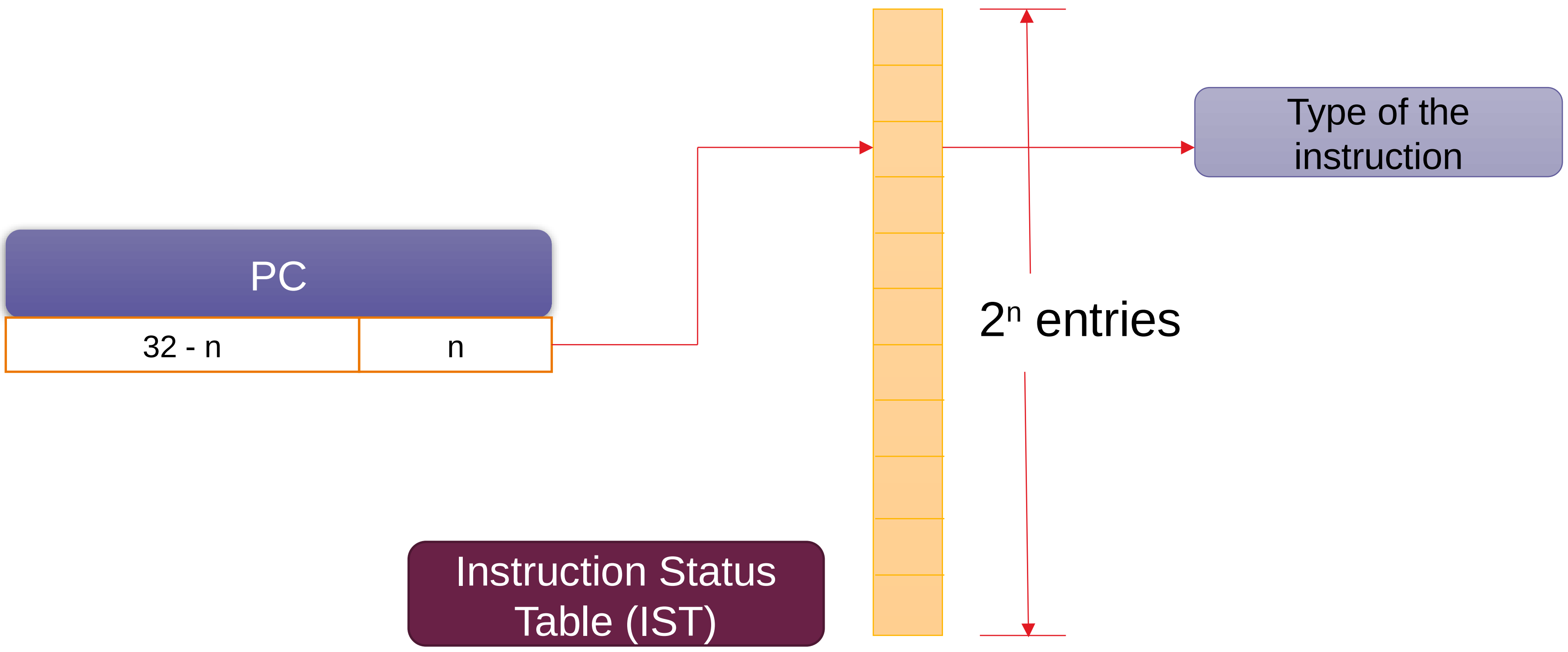
Insights

- Given a PC, the status of the instruction (branch or not) does not change.
- Can we use this information?

Approach

- The last time that we had seen a branch remember its PC
- Next time we see a PC, **check** if we have seen it before
- Also remember the type of the branch:
  - Unconditional branch
  - Conditional branch ▭ depends on the result of a previous compare instruction
  - Function call
  - Return

Make a structure in hardware to remember ...



Assumption Assume a 32-bit PC address

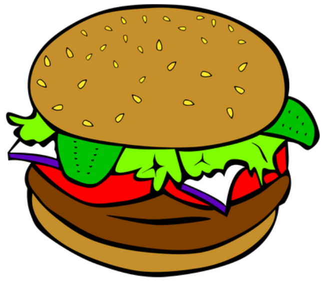
Basic Method of Operation

When we see a branch instruction

- Access its corresponding entry in the table
- Record the branch type

When we see an instruction:

- Check the corresponding entry of the table
- Read the type of the instruction



Why do we choose the *n* least significant bits?

Instruction Status Table (IST)

If we choose the least significant 10 bits of the PC address, we will have 1024 entries in the IST

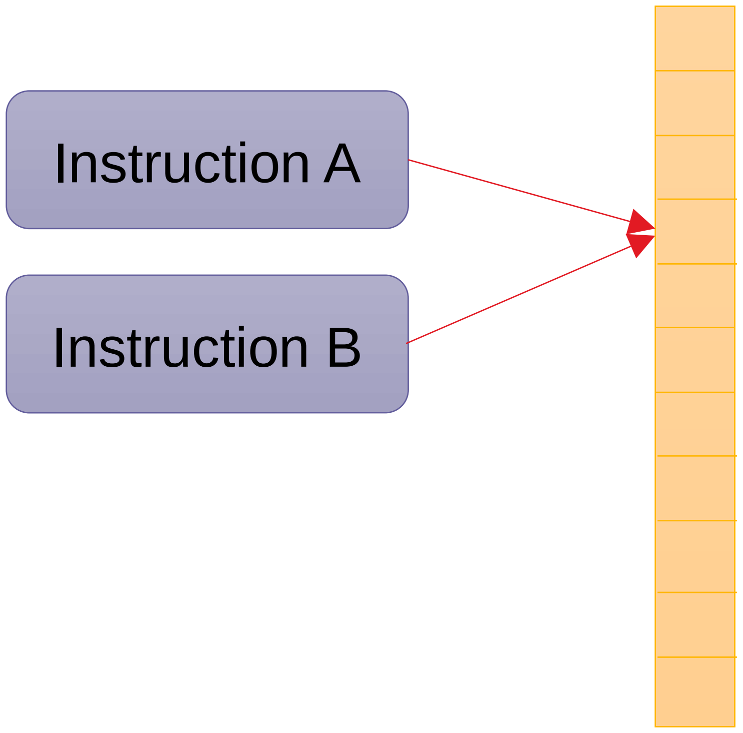
Why 10 bits?

For a 32-bit PC, we cannot have a 2^{32} entry table.

- Too big
- Too slow
- Too much of power
- Too much of area

We need to manage with a smaller table

Destructive Interference



Two instructions can map to the same entry; same last *n* bits.

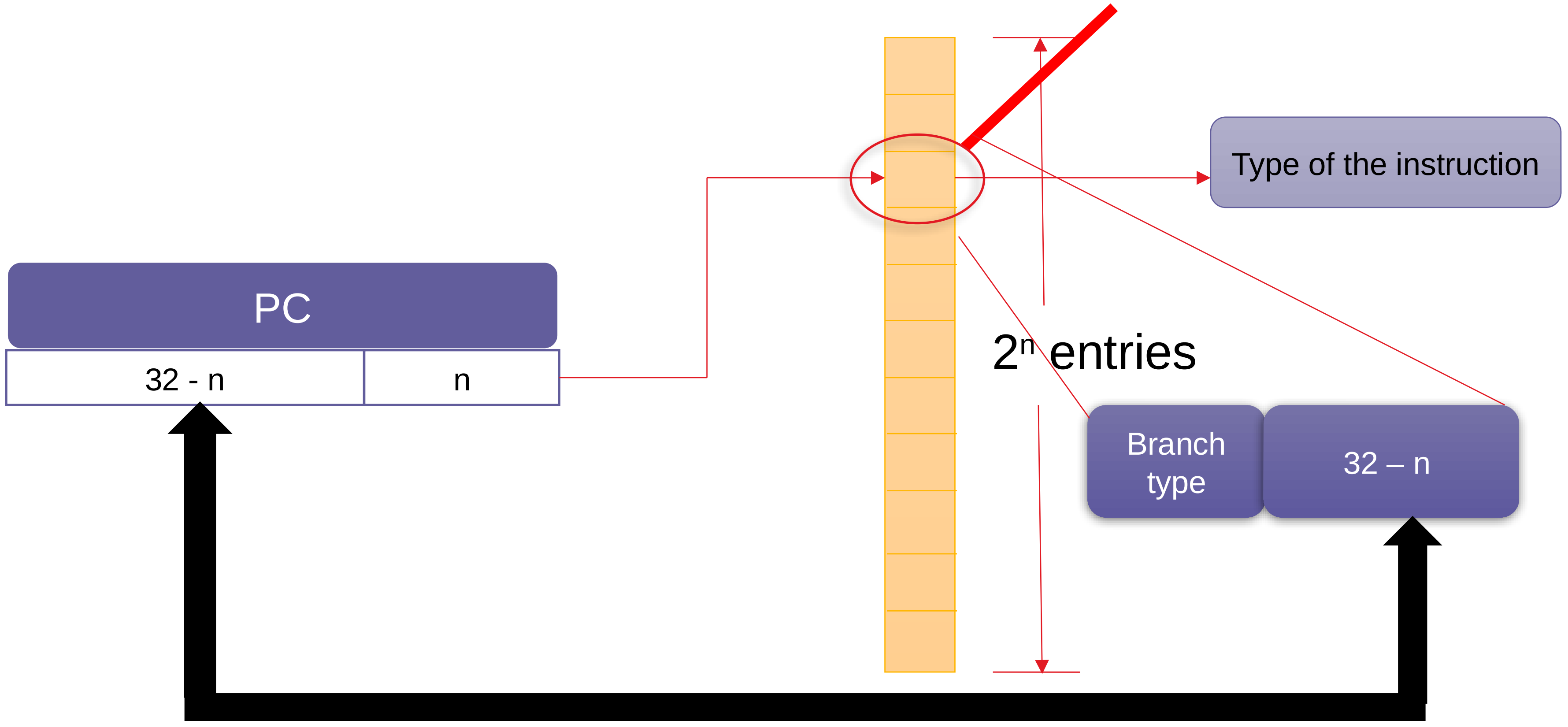
Defined as **destructive inteference**

Branch Aliasing

- Possible for a branch and a non-branch instruction to map to the same entry
- Possible for two branches to map to the same entry
- Possible for two non-branches to map to the same entry
- Need for disambiguation
  - Augment each entry of the IST

Disambiguation Between Addresses

Add a *(32-n)* bit tag



Disambiguation

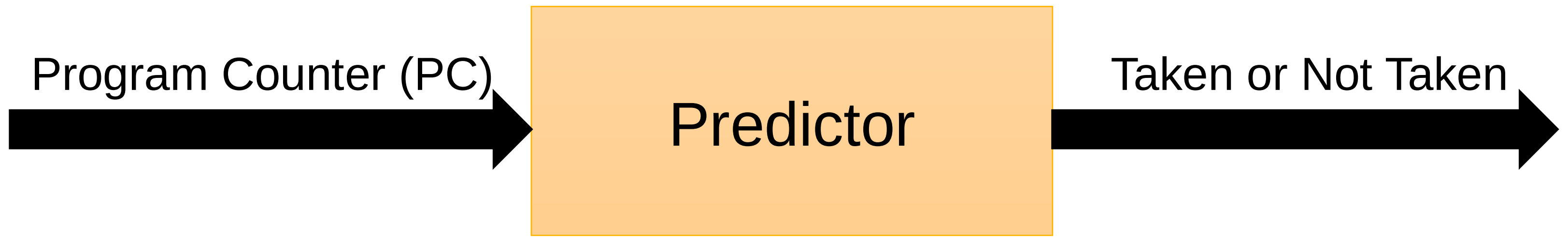
- We can keep the status of only branches in the IST
- However, for every instruction, we need to check the IST
- If there is no entry, then we need to predict that it is not a branch
- Can be wrong
- What to do ㅘ

Why does an IST work?

- Mainly because most pieces of code exhibit temporal locality
- In a given window of time instructions tend to repeat themselves

Contents

Predict the Direction of the Branch (Outcome)



- If a branch is unconditional ▭ no need to predict (taken)
- If a branch is a call/return ▭ no need to predict (taken)
- If a branch is conditional ▭ need to predict

Example Code

C

```
void foo() {
        for (i=0; i < 5; i++) {
                ...
        }
}
```

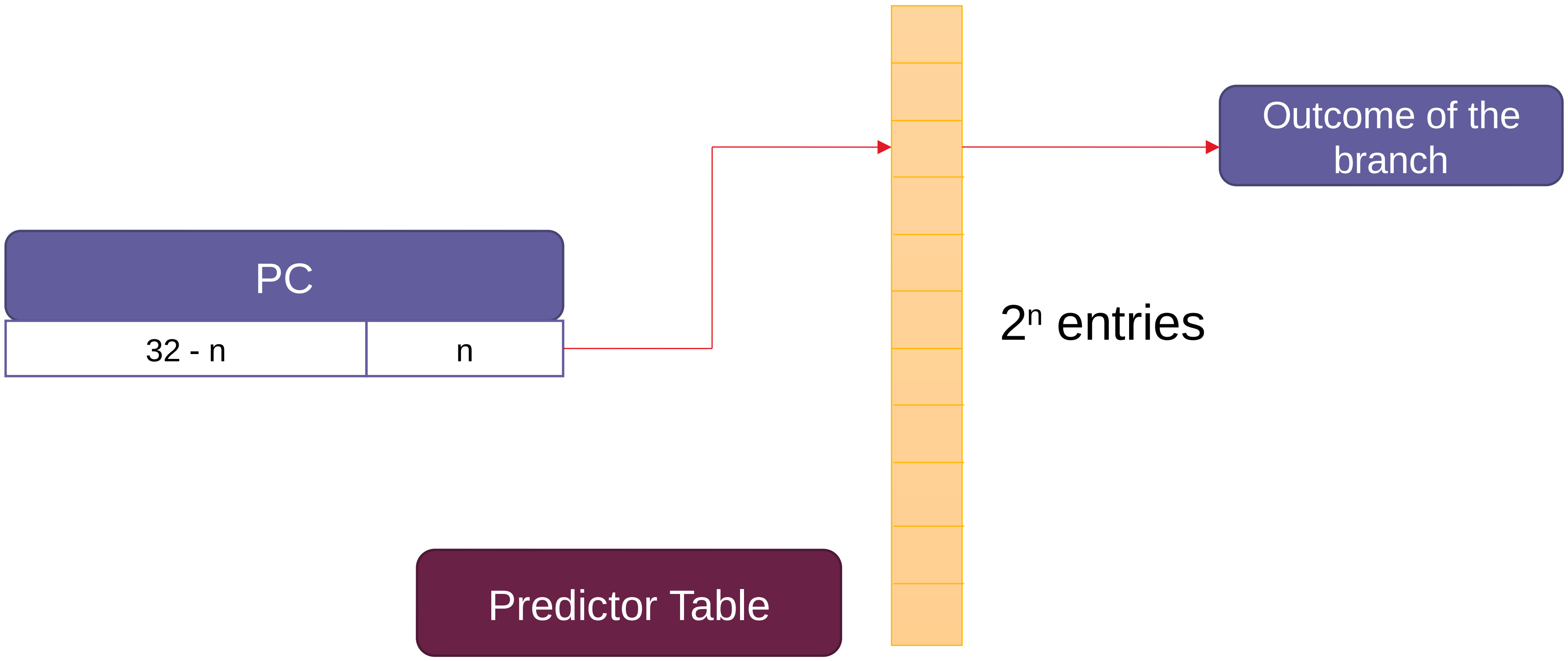
Assembly

```
.foo:
        mov   r0, 0
.loop:  cmp   r0, 5
        beq   .exit
        add   r0, r0, 1
        b .loop
```

Predict

This branch has roughly similar behavior most of the time. Predominantly not taken.

Simple Bimodal Predictor



Remember the last outcome.
Current prediction = last outcome

Bimodal Predictor - II

Each entry saves the last recorded outcome of the branch

- taken or not-taken

What is the problem:

- The *beq* instruction is evaluated 6 times
- Assume the default is not-taken
- First 5 times → correctly predicted not-taken
- 6th and last time → incorrectly predicted to be taken

Now assume we call the function foo() once again

- The first time → we will predict taken. This is wrong
- Again the last time will be wrong
- Misprediction rate: 2/6

Increasing Accuracy

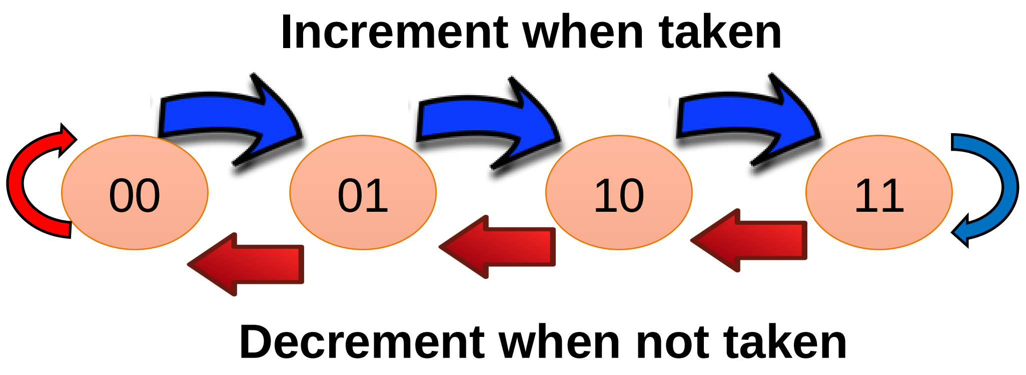
What is the problem?

- The *beq* instruction is fundamentally biased towards not-taken
- One exception at the end of the *for* loop cannot change its inherent behavior

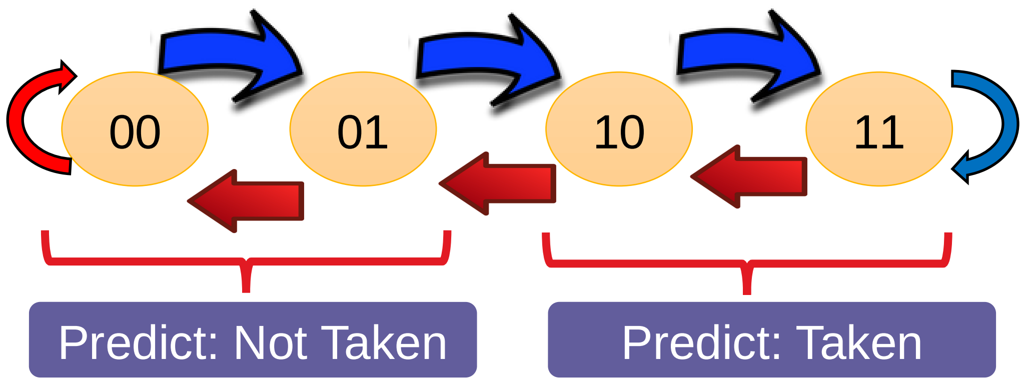
Let us add some hysteresis

- Instead of having a single bit in each entry, have a 2-bit counter

Saturating Counter



Saturating Counters



Algorithm for updates:

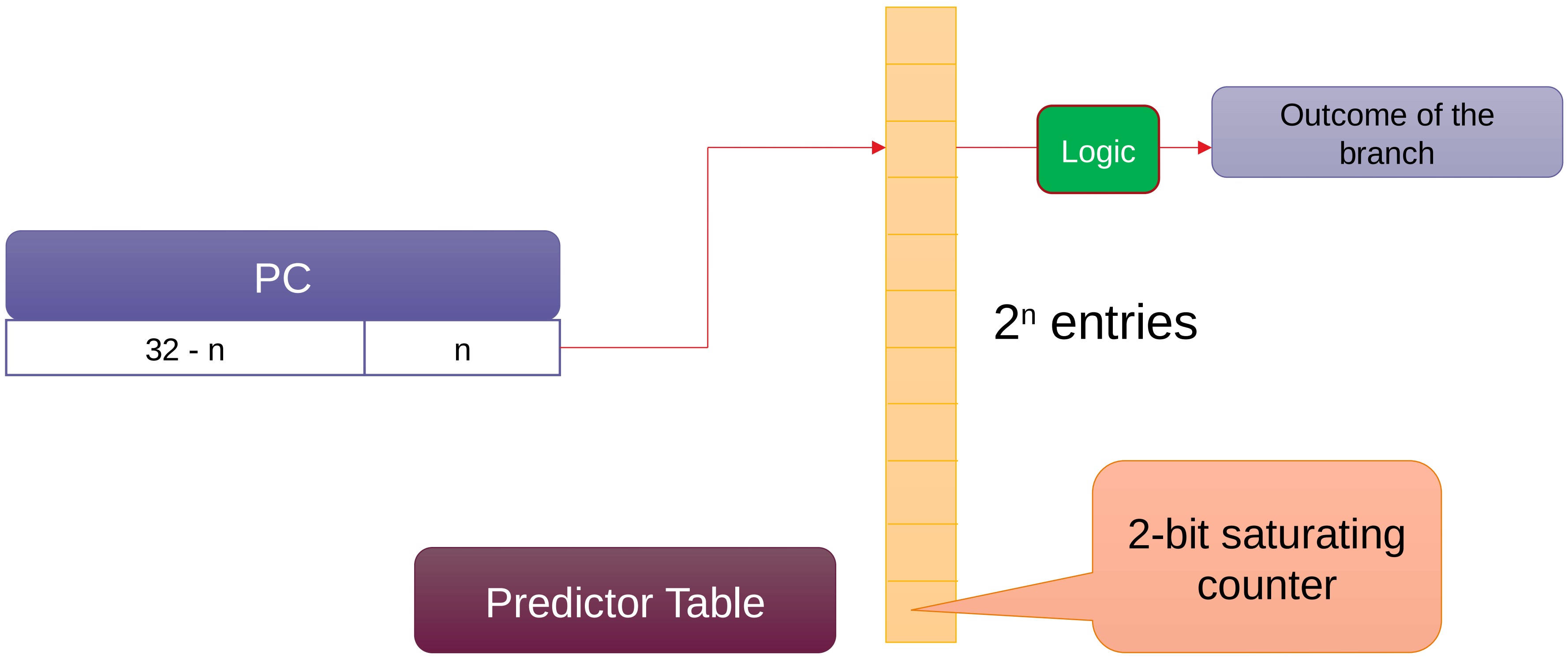
- If a branch is taken, increment the saturating counter
- If it is not taken, decrement the counter

For prediction:

- 00 and 01 ▭ not taken
- 10 and 11 ▭ taken

State	Name
00	Strongly not taken
01	Weakly not taken
10	Weakly taken
11	Strongly taken

Bimodal Predictor with Saturating Counters



Will it help?

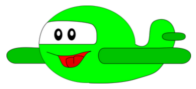
```
.foo:
          mov   r0, 0
.loop:    cmp   r0, 5
          beq   .exit
          add   r0, r0, 1
          b .loop
```

1st Time: Predict not-taken. Correct. Starts with 01. Moves to 00

2nd Time: Predict not-taken. Correct. Remains at 00

...

6th Time: Predict not-taken. Wrong. Starts with 00. Moves to 01



Misprediction rate: Down from 2/6 to 1/6

Why do saturating counters work?

- In this case, we have a degree of hysteresis. The status of the branch (*beq .exit*) moves between the strongly not taken and weakly not taken states.
- If a branch is strongly biased towards one direction, but once in a while changes its direction, we should use saturating counters.
- They capture stable behavior with occasional anomalies.

Can we do better?

What if we had?

C

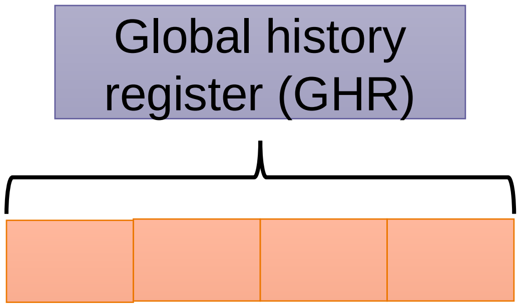
```
void foo() {
        for (i=0; i < 5; i++) {
                ...
                if (i == 4) {
                        foobar();
                }
        }
}
```

Assembly

```
.foo:
        mov   r0, 0
.loop:  cmp   r0, 5
        beq   .exit
        cmp   r0, 4
        bne   .inc
        call  .foobar

.inc:   add   r0, r0, 1
        b .loop
```

Global History Register (GHR)



- Let us have a shift register that records the history of the last *n* branches encountered by the processor
  - We have one bit for each branch (regardless of the PC)
- We record: 1 → taken branch, 0 → not taken branch
- Let us consider a 2-bit shift register also known as the GHR
  - GHR → Global History Register
- We have two conditional branches in the running example
  - *beq .exit*
  - *bne .inc*

Status of the GHR

Beginning of the 2nd Iteration

- 01

Beginning of the 4th Iteration

- 01

Beginning of the 5th iteration

- 01

Beginning of the 6th iteration

- 00

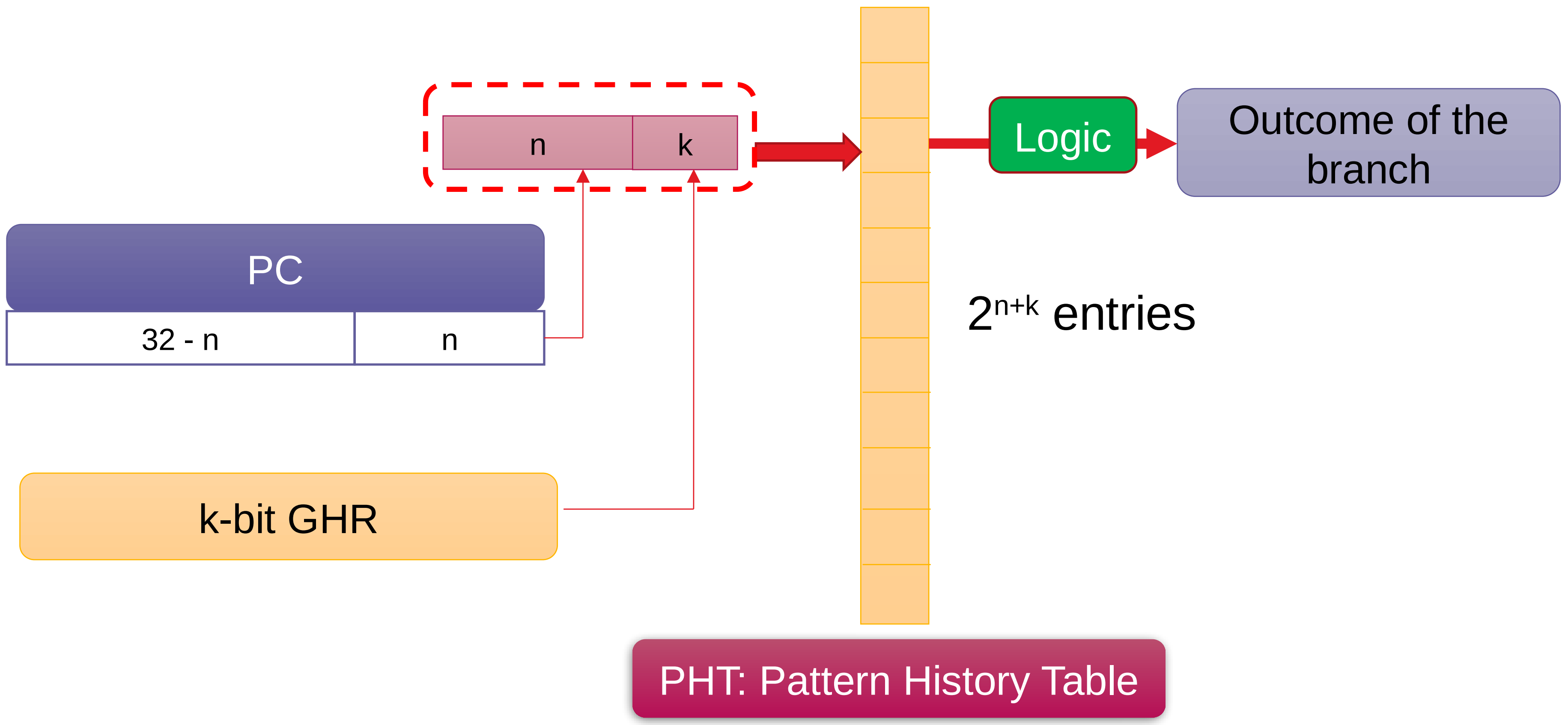
```
.foo:
            mov   r0, 0
.loop:      cmp   r0, 5
            beq   .exit
            cmp   r0, 4
            bne   .inc
            call  .foobar

.inc:   add   r0, r0, 1

            b .loop
```

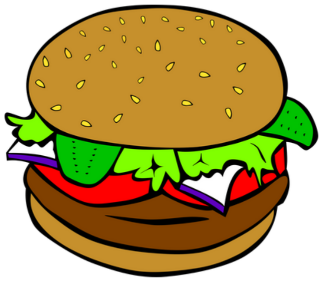
Use the GHR information to also decide the direction of the branch.

GAp Predictor



GAp Predictor - II

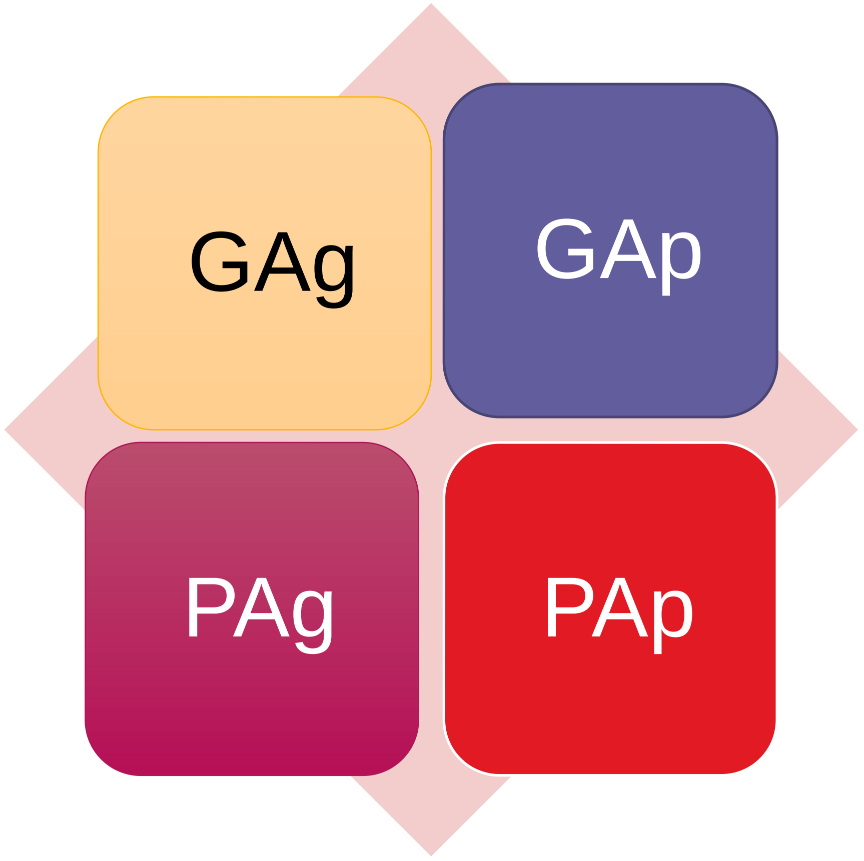
- We create 2^k times more entries in the predictor table
- Capture the global branch history
- Use both the global history as well as local (per PC) history to make the prediction.
- The accuracy is expected to increase.
  - The branch in the last iteration was predicted correctly
- In general we can create different combinations of:
  - The PC bits and the GHR's branch history



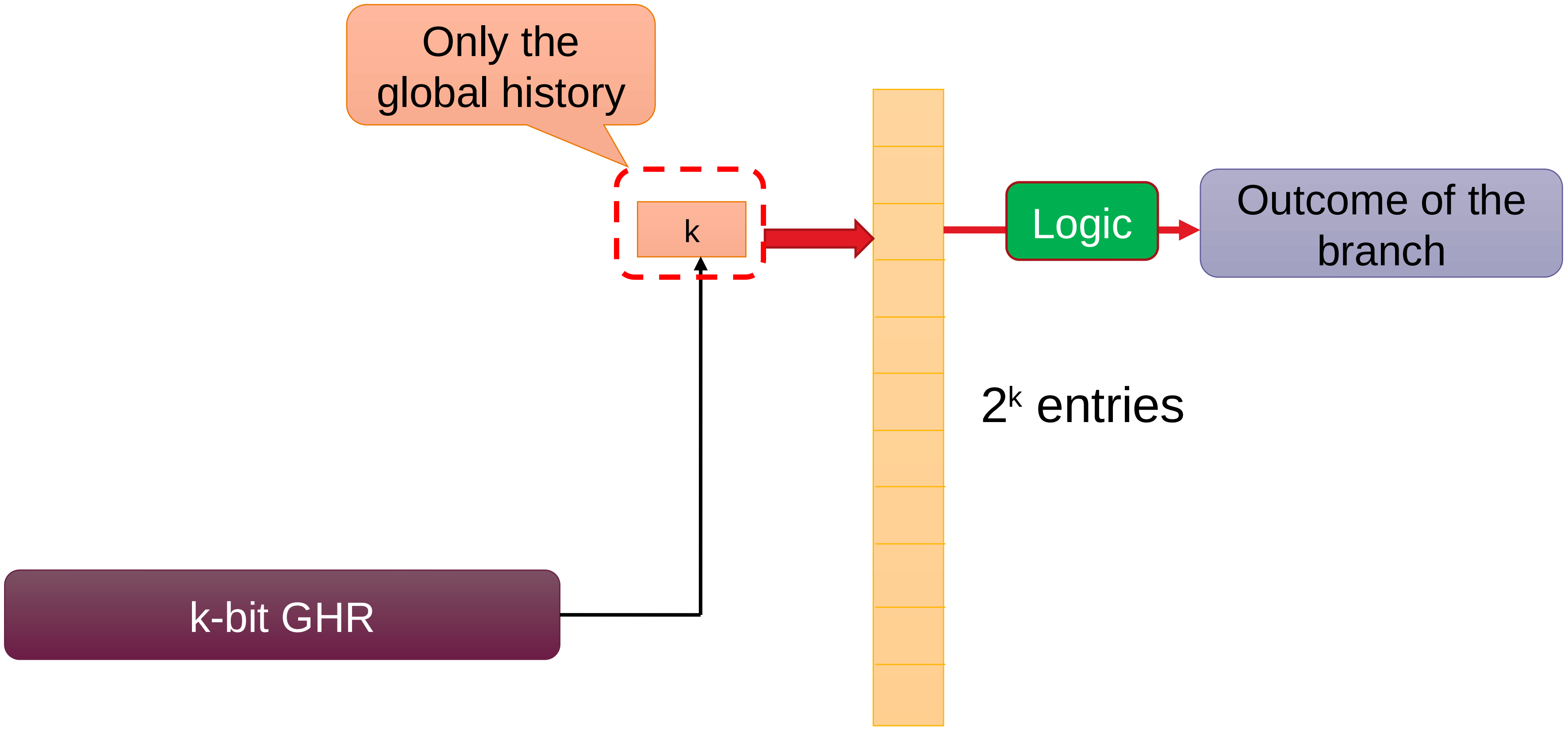
We have a branch that has an alternating pattern: taken, not-taken, taken, ... Will a GAp predictor capture the pattern?

Can we design a class of predictors?

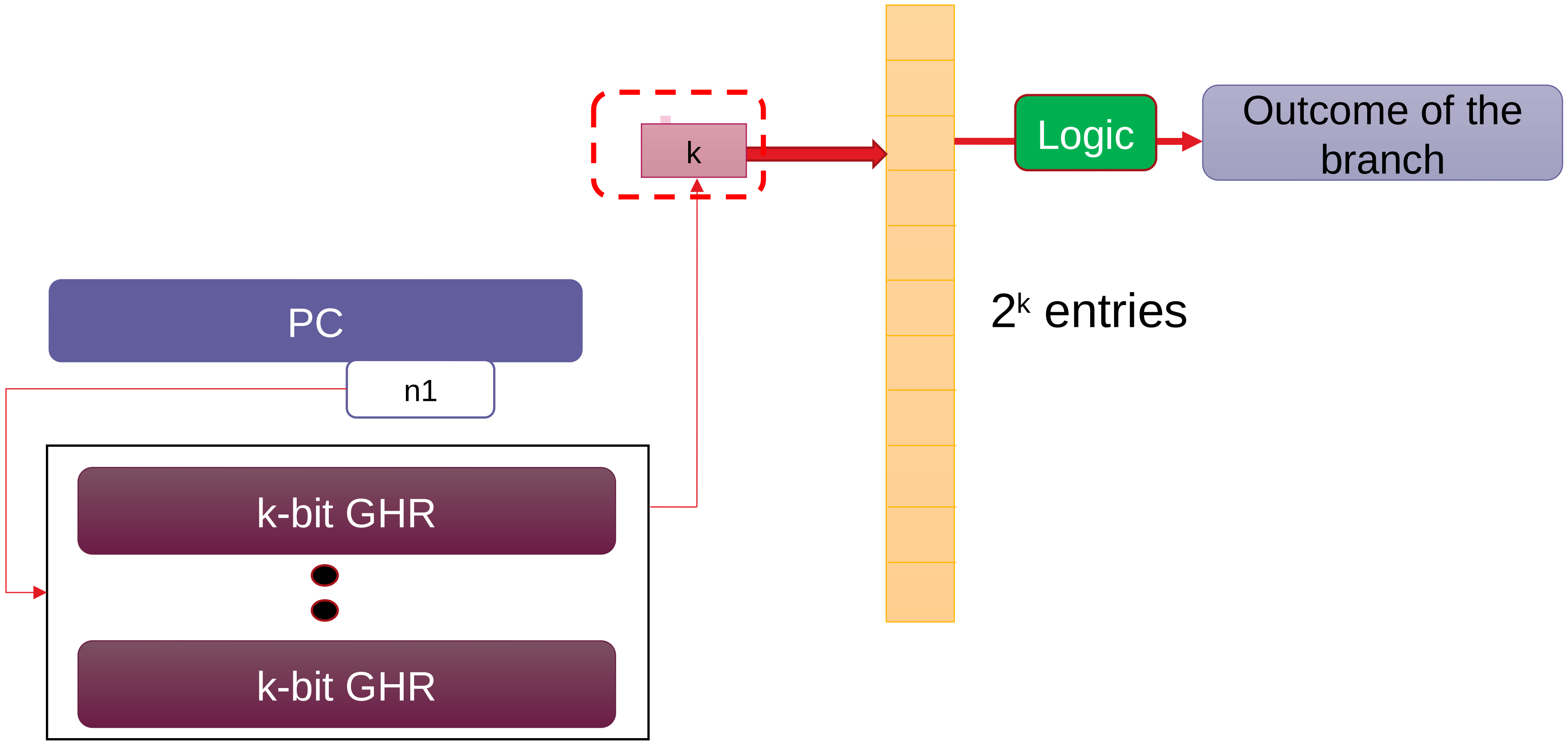
G □ Global history, P □ Per PC pattern history



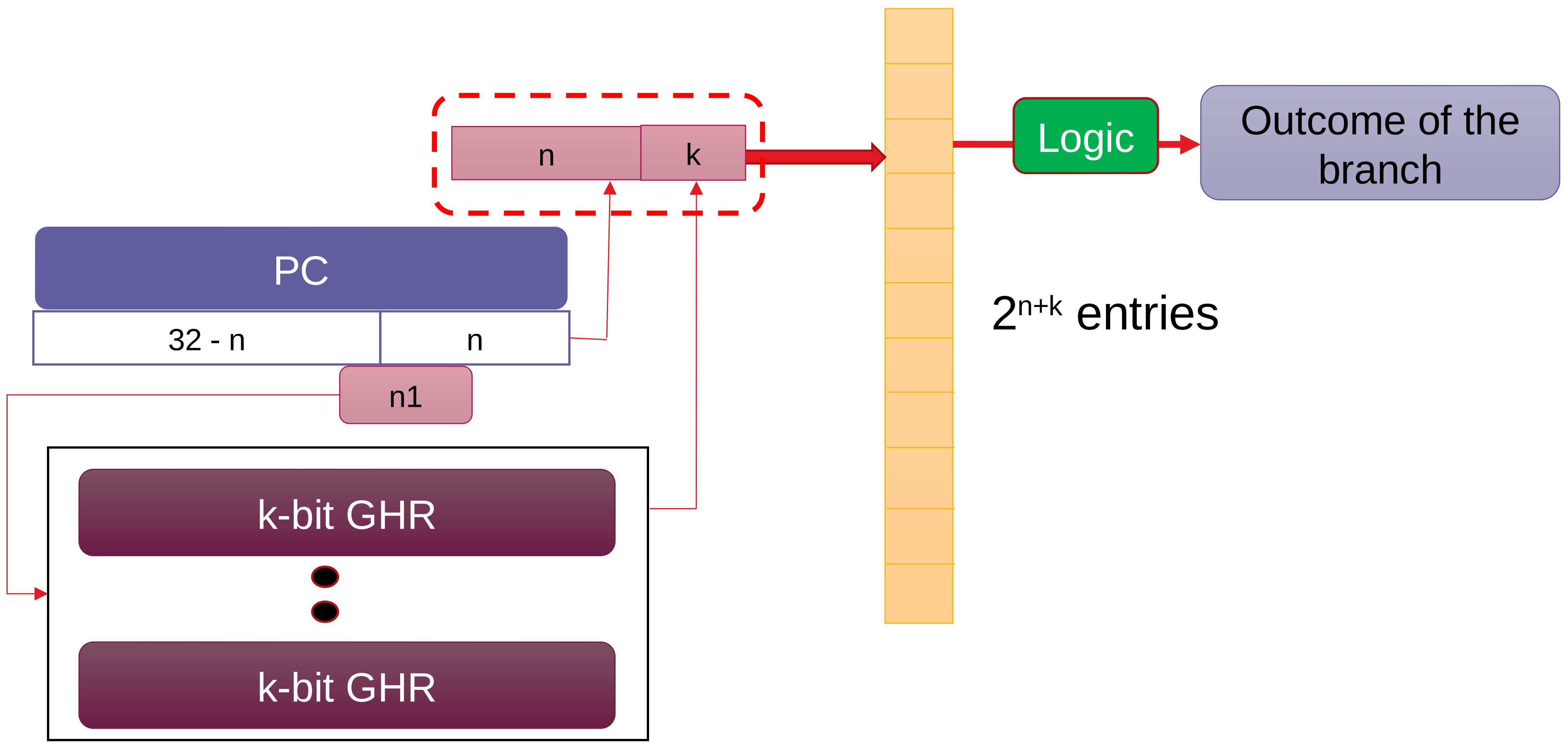
GAg Predictor



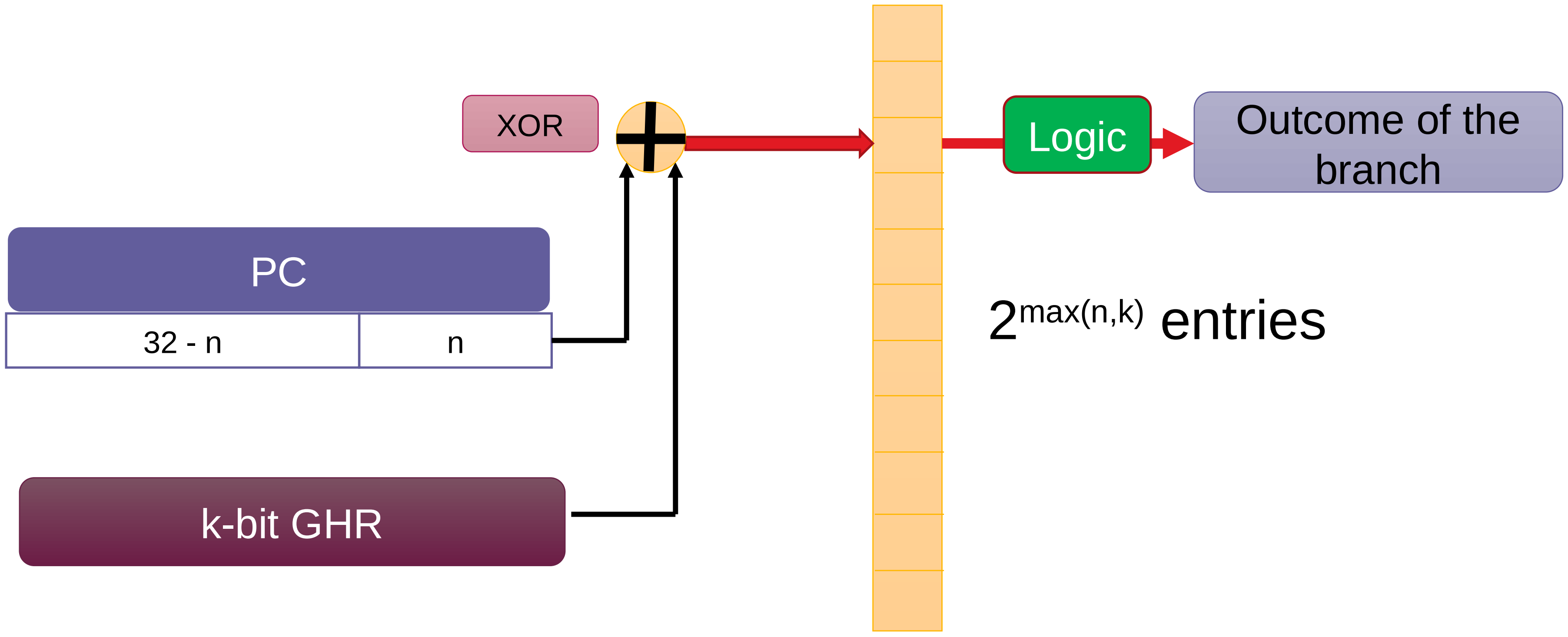
PAg predictor



PAp predictor



Another way of combining information: GShare

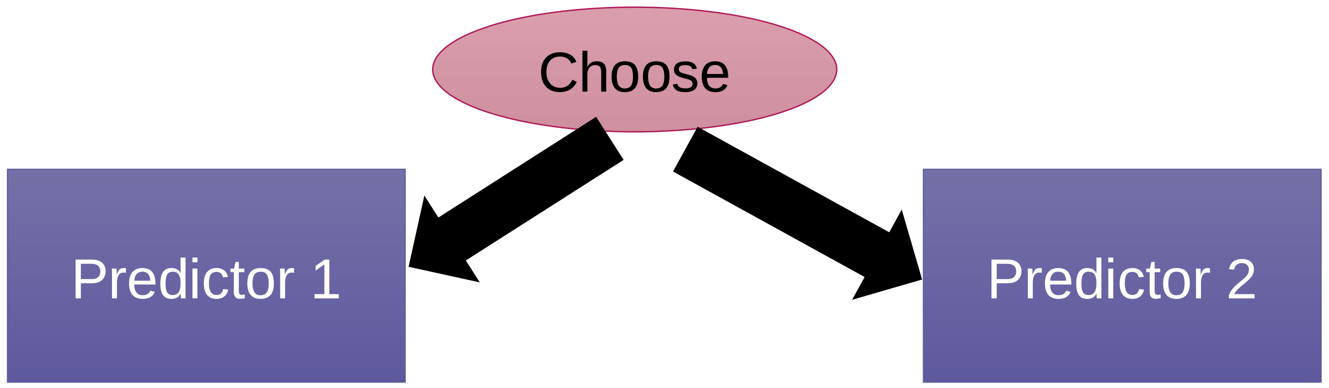


Tournament Predictor

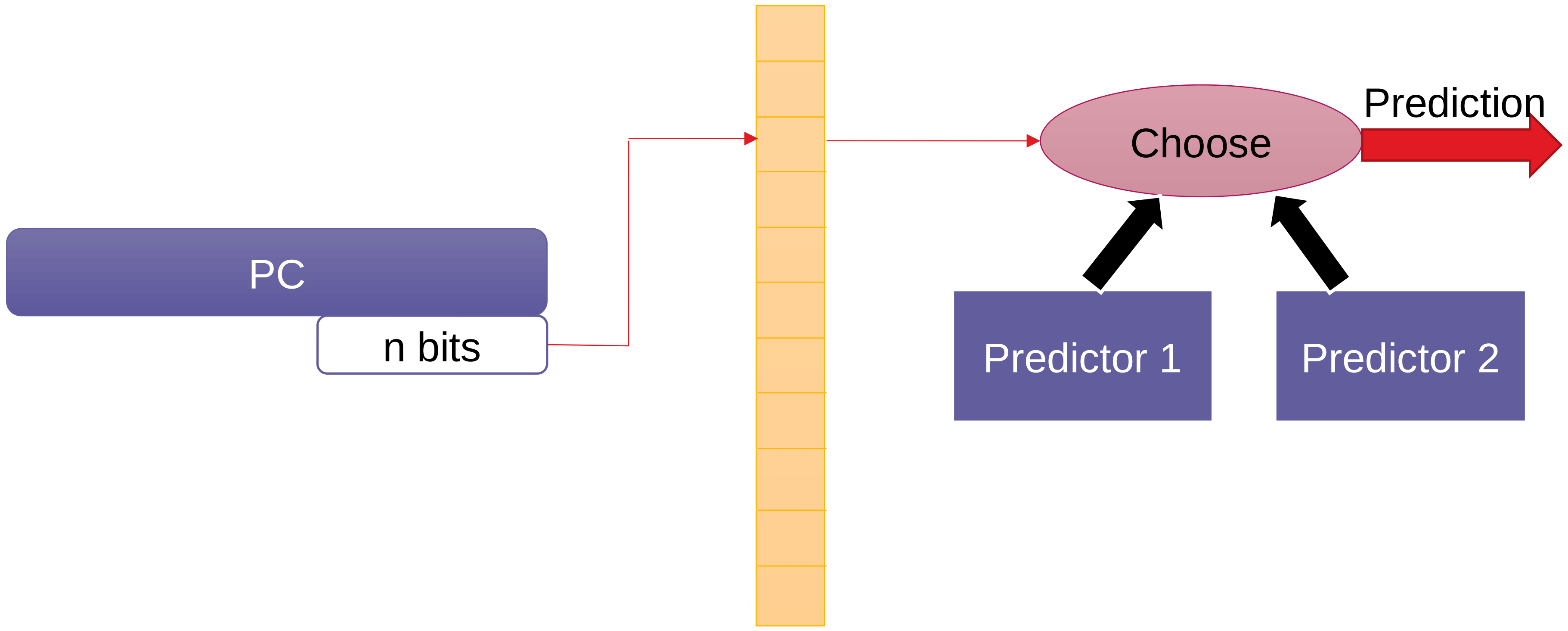
- Different predictors have different accuracies for different program snippets
- For a very biased branch a bimodal predictor with saturating counters is the best
- For an alternating pattern, a PAp predictor works very well
- How to know which predictor is the best for which piece of code?

What to do?

- Answer: Use two predictors, and choose between them



Tournament Predictor



Operation of a Tournament Predictor

Prediction

- Find the entry in the choice array
- Choose the predictor based on the value of a saturating counter
- Use its prediction ▭ because of the saturating counter, we automatically choose the predictor that performs the best for a given PC.

Training

- Train both the predictors
- Train the entry in the choice array if we chose the wrong predictor
- If both the predictions are the same, we don’t modify the choice array
- If they differ, we increment the counter if Predictor 1 was correct and decrement it if it was incorrect.

Other methods of prediction

Reduce aliasing

- Incorporate a few tag bits in each entry of the predictor
- Have multiple predictors for different subsets of branches
- Include a bias bit with every branch (its most likely direction), and just predict if we need to agree with the bias bit or not (agree predictor)

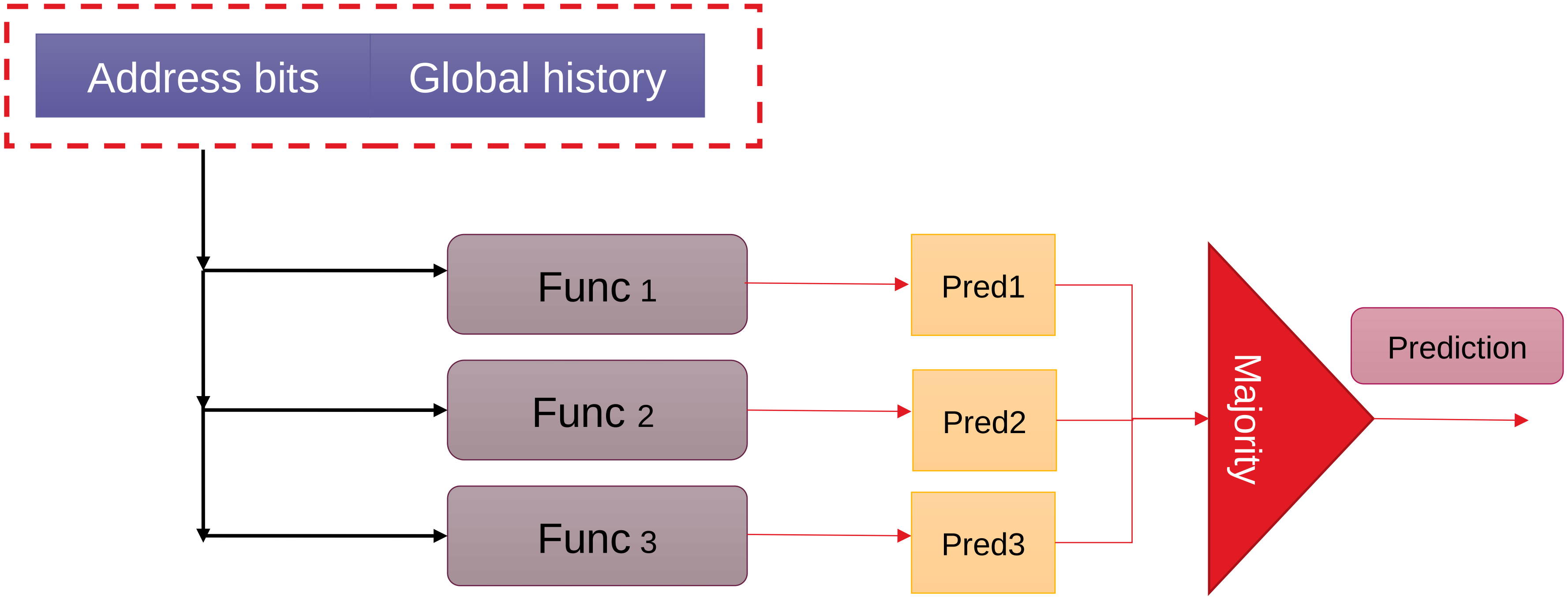
Better use of bits

- Separate high confidence and low confidence branches. Dedicate more bits to low confidence branches

Examples of other predictors:

- Bi-mode, Agree, Skew, YAGS, TAGE

Skewed Branch Predictor

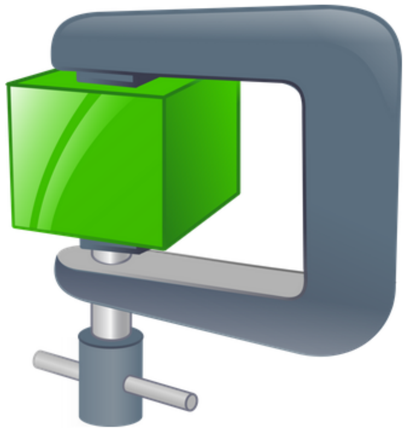


Prediction and Compression

- Consider a sequence of <PC,outcome> pairs
- Let's compress this sequence using a standard program: zip, rar
- Is the prediction accuracy related to the compression ratio (size of compressed file/ size of original file)?
- **YES**

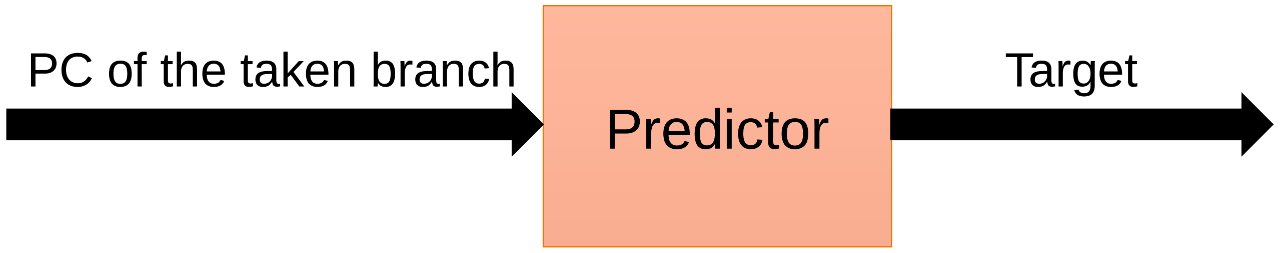
Take a course on information theory

- Read about the Fano's inequality

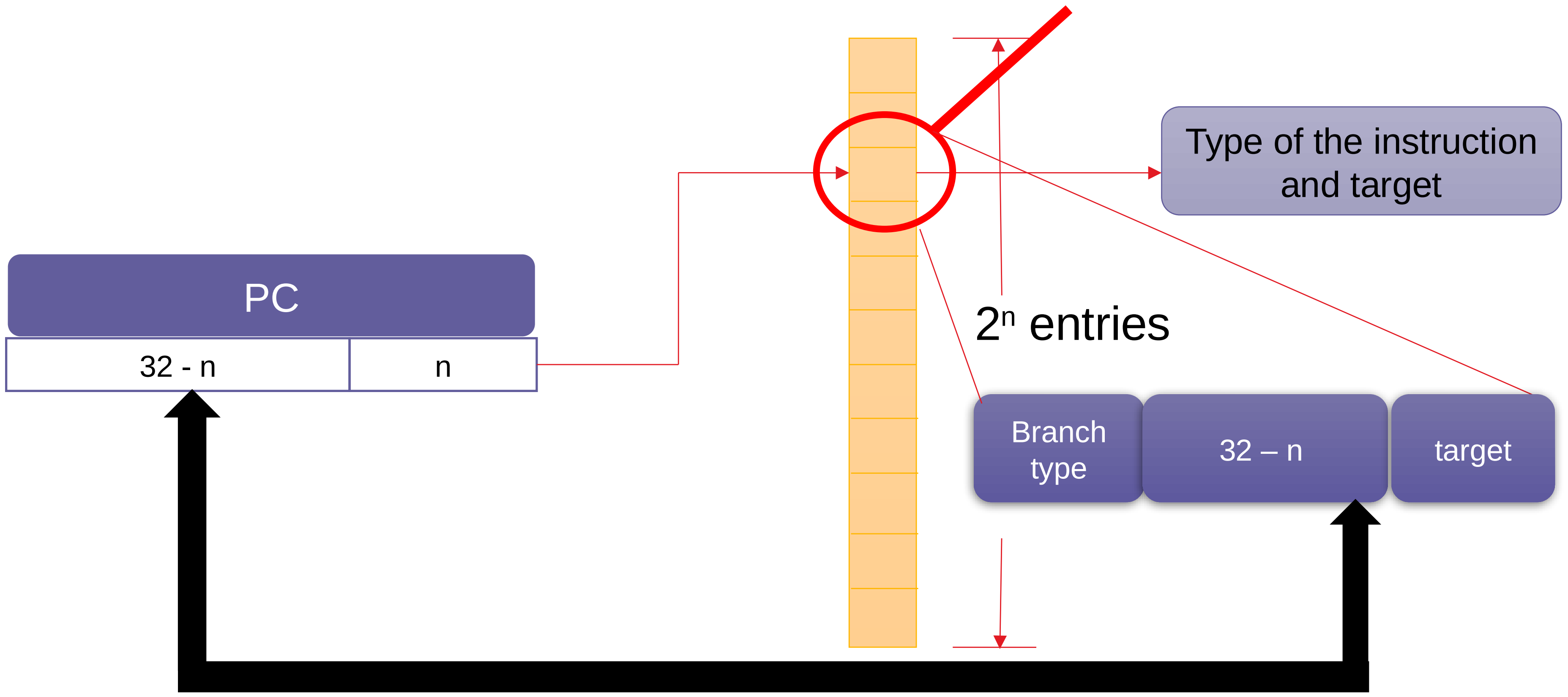


Contents

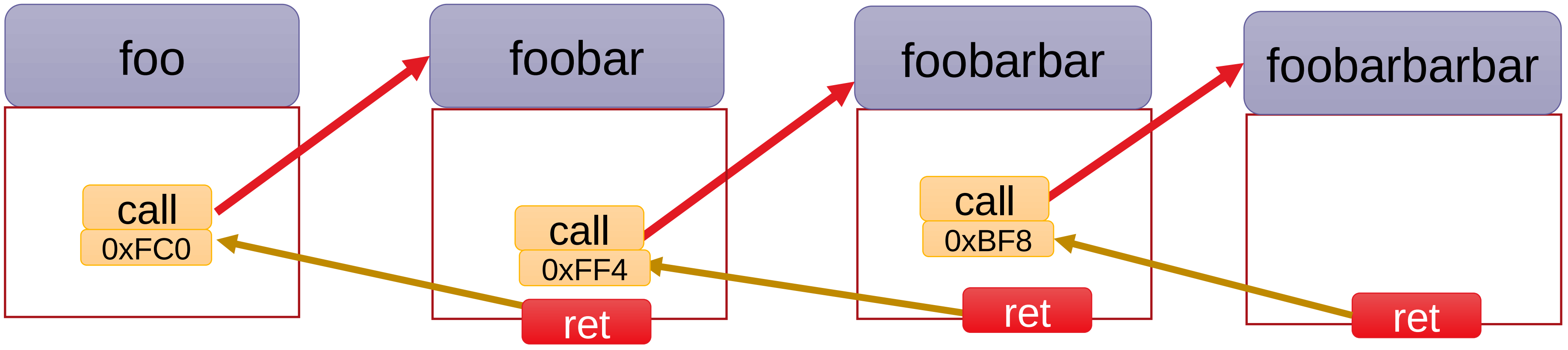
Predict the Branch Target



Use the IST. Let us call it the Branch Target Buffer (BTB)



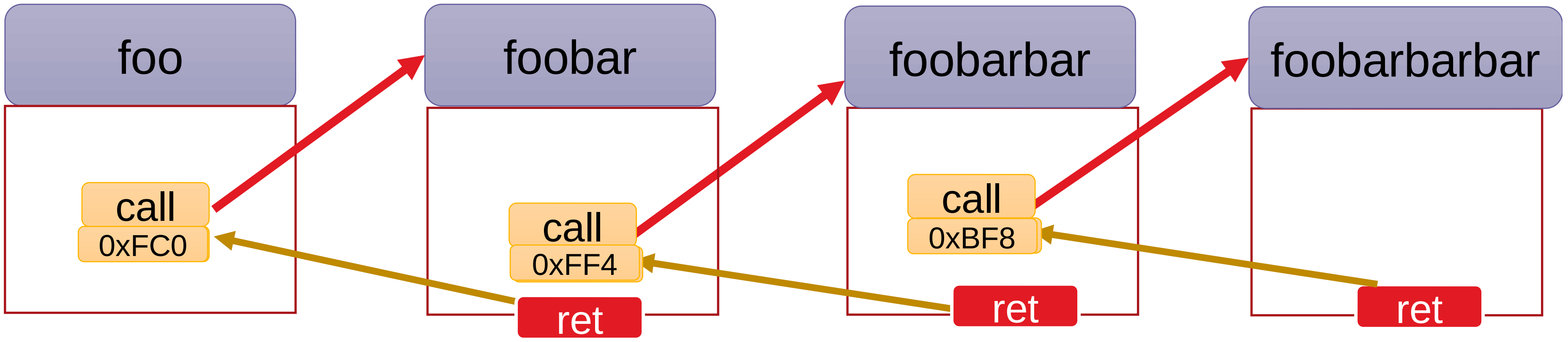
Calls and Returns



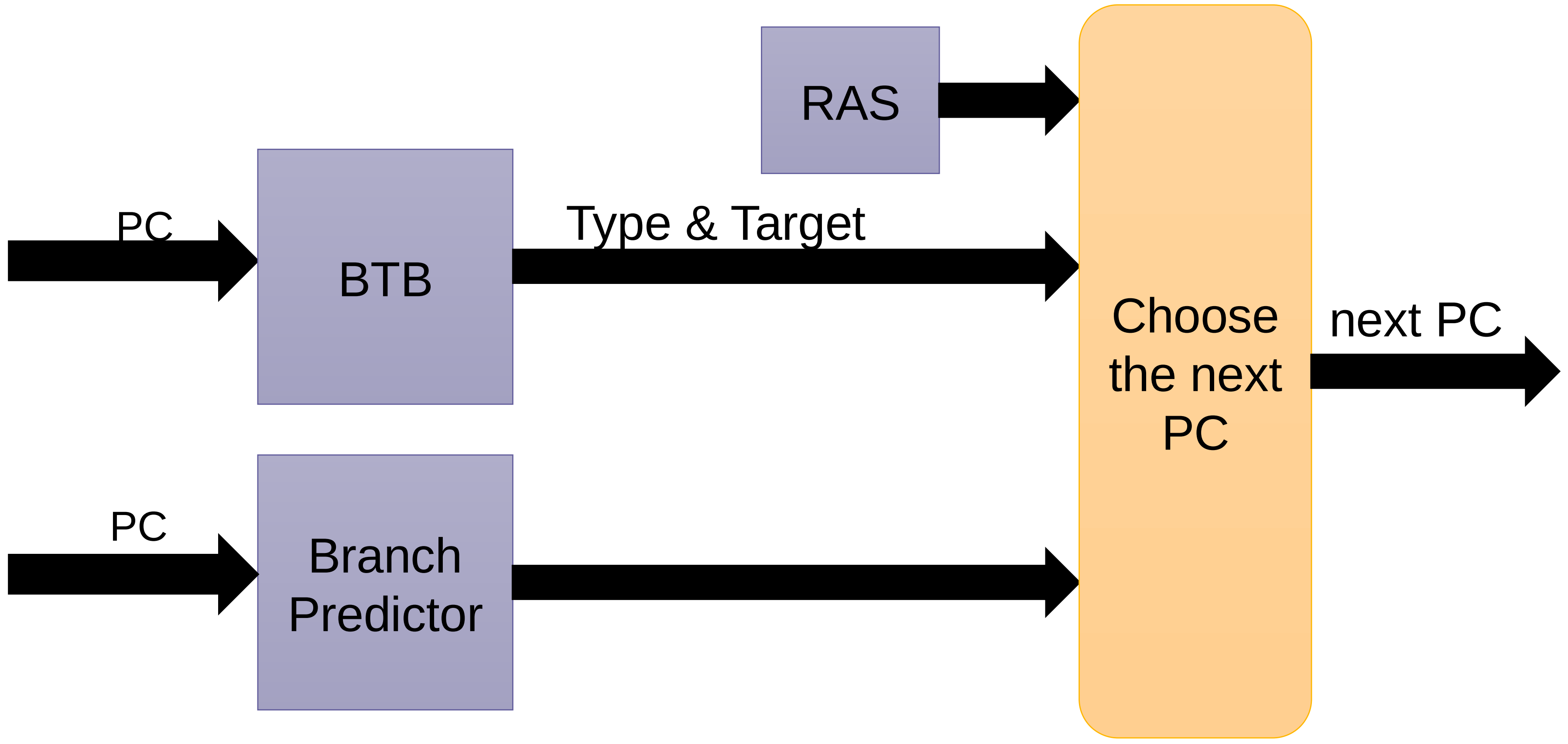
How to predict return addresses?

Solution: Use a stack

Return address stack (RAS)



Summary: The Branch Prediction System

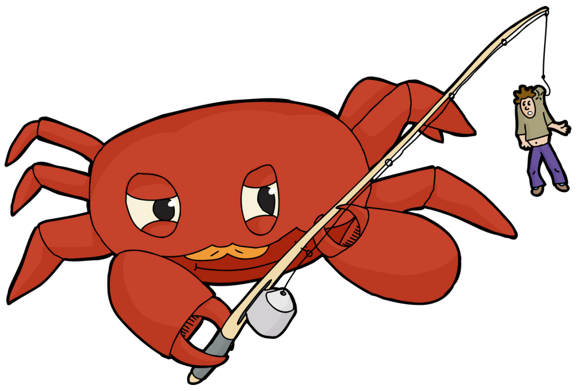


Contents

The Process of Decoding

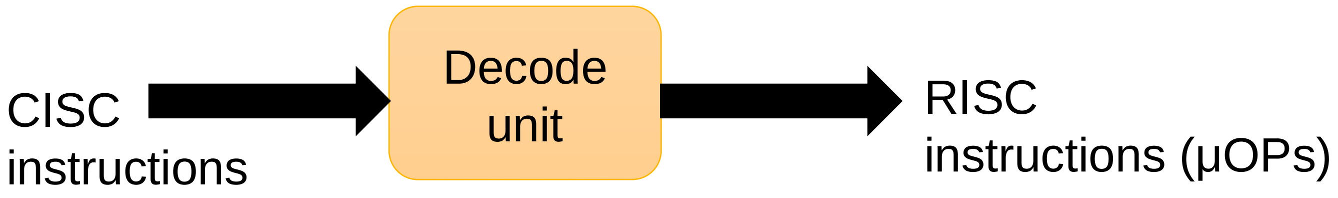
- Expand all the immediate values to 32 or 64-bit values
- Extract all the fields
- Compute the branch target (if branch is taken)
- Add all implicit sources (such as for the return instruction)
- Create the instruction packet

Issues with CISC Instructions



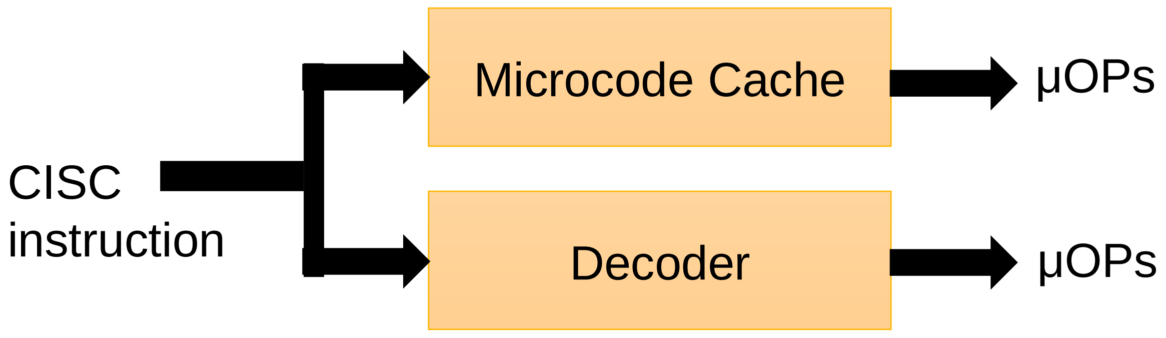
CISC Instructions

- Do not have a fixed length
- In x86 the length varies from 1 to 15 bytes
- It is hard to fetch multiple instructions at once
  - Need to know the boundaries of instructions
  - It is hard for OOO pipelines to process them
    - Most CISC processors internally convert from CISC to RISC

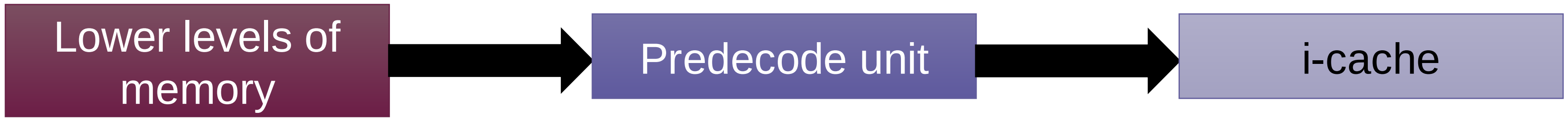


Regular Decoder vs Microcode Cache

- x86 has some very complex instructions
- Consider the *rep movsd* instruction
  - It can be used to copy *n* elements from one location to the other ▭ all in one single instruction
  - Such instructions map to a large number of µOPs
- In comparison, simple add and sub instructions with register operands map to a single µOP.

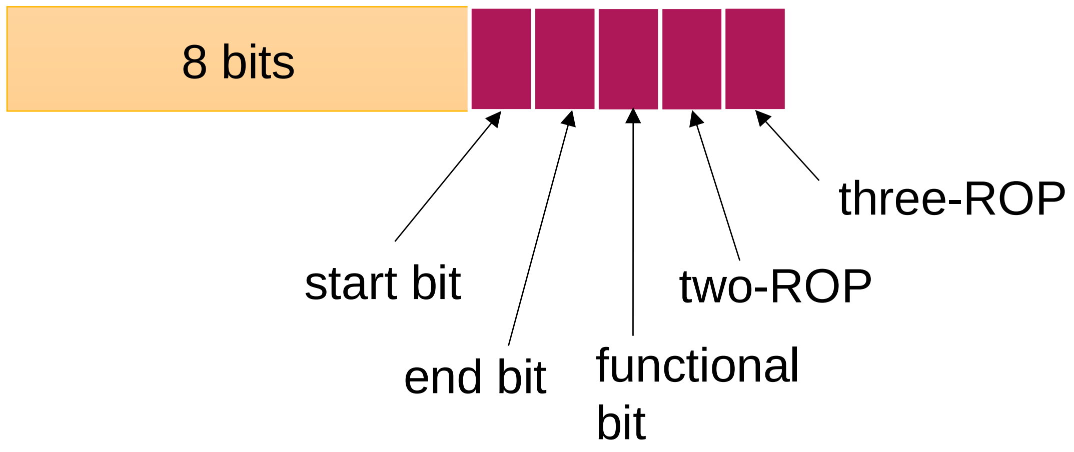


Predecoding CISC Instructions



- When we read in an instruction into the i-cache
  - We predecode it
  - We annotate each byte with additional information
  - When we read the bytes from the i-cache, we can use this additional information to decide the instruction boundaries. [Narayan and Tran, 1999]

Predecoding - II



- Start bit ▭ Starting byte of an instruction
- End bit ▭ Last byte of an instruction
- Functional bit ▭ Interpretation depends on the implementation.
- two-ROP and three-ROP bits ▭ Number of μOPs in an instruction

Optimizing Operations on the Stack Pointer

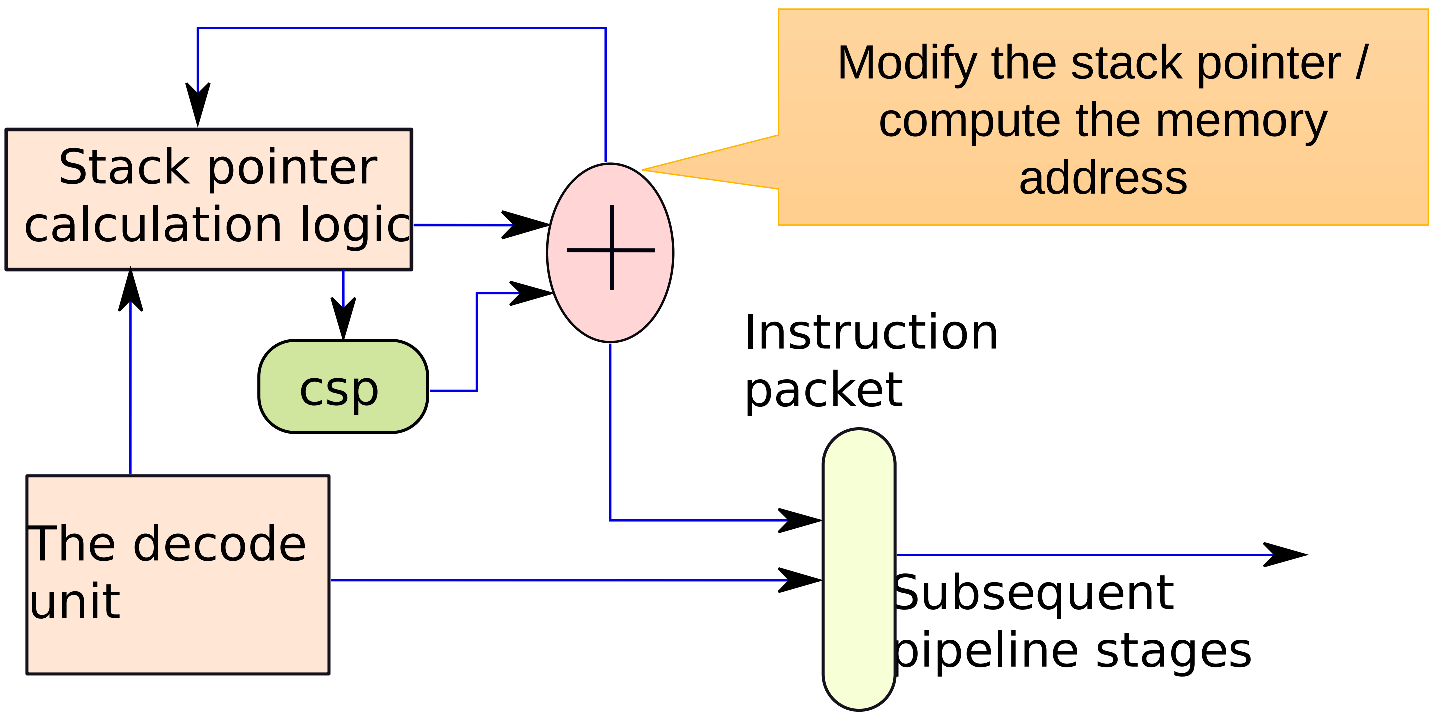
Example code

```
sub sp, sp 8
st r1, 0[sp]
st r2, 4[sp]
....
ld r2, 4[sp]
ld r1, 0[sp]
add sp, sp, 8
```

- There is a pattern here.
  - We either modify the stack pointer by adding or subtracting a constant
  - The load and store instructions access the stack pointer.
  - Update the stack pointer locally or get the memory address directly

Have a copy of the *sp* register in the decode stage itself.

Add a *csp* register in the decode stage


Modify the stack pointer / compute the memory address
Stack pointer calculation logic
csp
The decode unit
Instruction packet
Subsequent pipeline stages

Corner Cases

ld sp, 12[r1]

- When we encounter such instructions, we set the value of *csp* to null.
    - Treat *sp* as a regular register in the OOO pipeline.
    - Accumulate the difference, Δ, between the value returned by the load addr and the current value of the stack pointer.
    - When the load instruction returns with its value: set

$$csp = addr + \Delta$$

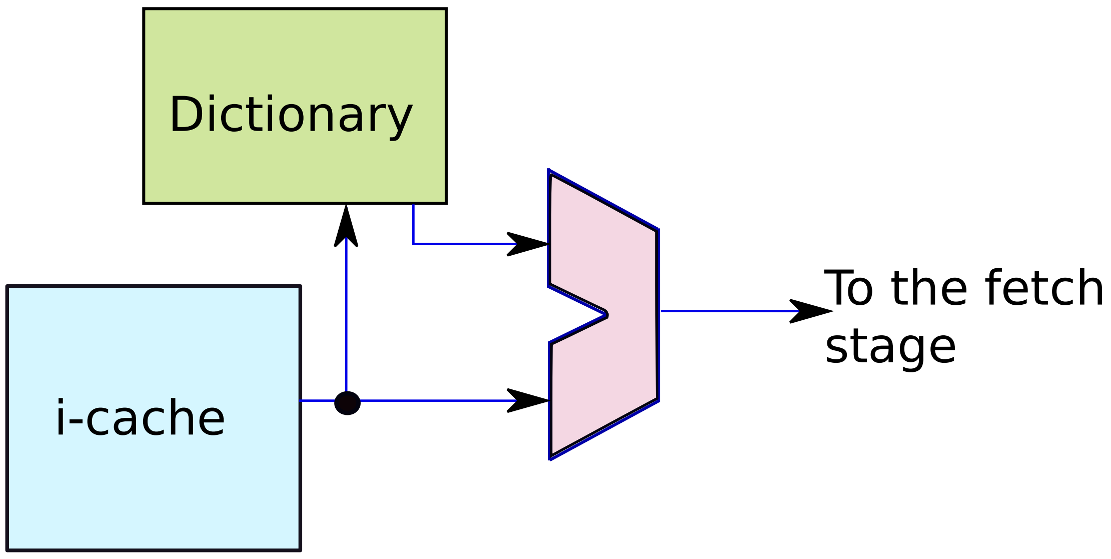
Advantages

- Load instructions to the stack can be issued early because we compute the address at decode time.
- For load and store instructions to the stack, the address need not be computed in the pipeline.
- We can get rid of many instructions that update the stack pointer. This will effectively reduce the dynamic instruction count in the rest of the pipeline.

Instruction Compression

- Instruction caches have a limited size: typically 32 to 64 KB
- The performance is extremely sensitive to their size
- Hence, we wish to pack as many instructions as possible
- **Approach 1**: Reduced-width instructions
  - Support a limited number of opcodes
  - Avoid encoding complicated flags and options
  - Reduced view of architectural registers
  - Reduce the size of the immediate fields
  - Implicit operand (accumulators):
    add r1, r2 (r1 = r1 + r2)

Instruction Compression



- The compiler or profiler identifies sequences of frequently executed instructions
- Each sequence is replaced with a code word
- The code word uses fewer bits
- The code word ▭ sequence mapping is stored in a separate hardware structure: dictionary
- Reduces code size. We can even store decoded instructions

Conclusion

e can use past history to predict if a PC is
branch or not.

Saturated counters are used to capture steady-
state behavior that can have occasional
anomalies.

Often, the context determines the direction of branches.

global history predictor such as GAp or PAp is useful in
this case.

he BTB records the branch type and the target.

ecode time optimizations are required to
ncrease performance.

The End