3
The Fetch and Decode Stages

The aim of this chapter is to design a fetch engine that has a very high bandwidth. This means that we can supply as many instructions as possible in a single cycle to the rest of the pipeline. A high bandwidth fetch engine can typically supply 4 or 6 instructions per cycle to the rest of the pipeline. If the fetch stage becomes a bottleneck, then the entire OOO processor will become very slow because irrespective of the speed of other stages, they will simply not have enough instructions to work with.

Designing a high bandwidth fetch engine is a classical problem in computer architecture and as of 2020 this field has matured. Let us take a brief look at the components of a modern fetch engine (see Figure 3.1).

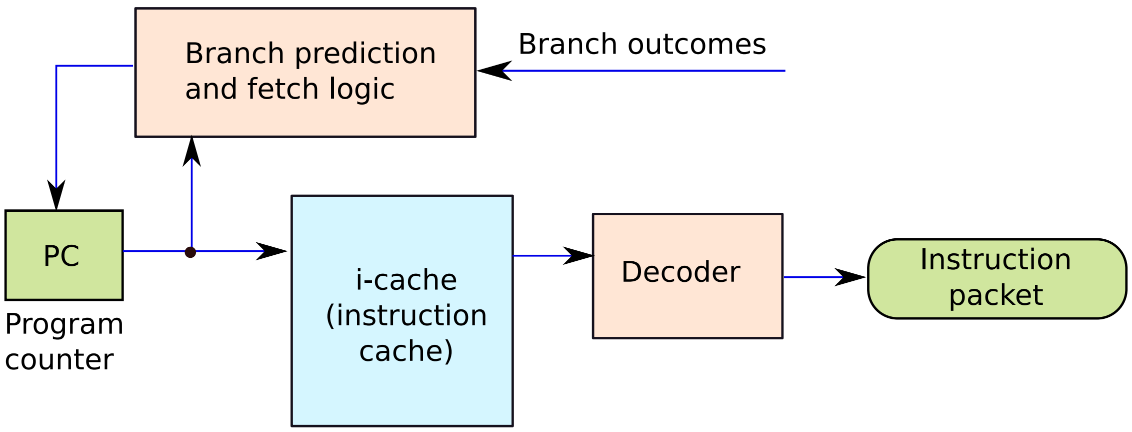
Branch prediction and fetch logic
Branch outcomes
PC
Program counter
i-cache (instruction cache)
Decoder
Instruction packet

Figure 3.1: Fetch and decode stages in a modern pipeline

The components shown in Figure 3.1 are the instruction cache (i-cache), fetch logic, and the branch prediction unit. They send the fetched instructions to the decoder. The fetch engine reads its inputs from the instruction cache. The final output of the fetch and decode process is a set of instructions that have been decoded. Recall that the process of decoding extracts the information encoded in an instruction. This information includes register ids, immediates (constants), program counter based displacements for branches, and special instruction flags. The decode unit adds additional information to control the multiplexers along the data path such that the instruction can be executed properly by the execution units, and its results can be written back to either the data memory or the register file. The bundle of information that the decoder generates for each instruction is known as the *instruction packet*.

Definition 14
An instruction packet is defined as the bundle of information that is required to process an instruction in all the stages of a pipeline. It is generated in the decode stage and typically contains the ids of the source and destination registers, the values of constants embedded in the instruction (immediates), branch and memory offsets, special flags, and signals to control execution units throughout the pipeline.

3.1 Instruction Delivery from the I-Cache

The basic logic in the fetch engine of an OOO pipeline is no different from that of an in-order pipeline. If the size of each instruction is 4 bytes, then we need to fetch 4 bytes starting from the program counter (referred to as PC[1]). The addresses of the 4 bytes are in the range: PC, $PC + 3$. The next instruction starts at the address (PC + 4). Now, if we need to supply the pipeline with 4 instructions, we need to fetch 16 bytes from the i-cache.

Recall that a cache is a memory structure that contains data and instructions within the chip. We shall describe the internal details of caches in Chapter 7. For now, we can assume that a cache is a linear array of cache lines or cache blocks (the term *line* and *block* will be used interchangeably). A cache line or cache block is an atomic unit of data in a cache. It is read and written in one go. The typical instruction cache line size as of today is either 32 bytes or 64 bytes. The reason for using a cache is that these are fast memory structures unlike off-chip main memory that is very slow. We can quickly read data from such caches. Additionally, they can sustain a very high read throughput.

Let us now devise a mechanism to fetch these 16 bytes. Assume the simpler case first where there are no branches among these four instructions. In this case, we need to read 16 contiguous bytes from the i-cache.

Instruction caches are organised as regular caches with a block (line) size that is typically in the range of 32 to 64 bytes. Thus, each line contains 8 to 16 4-byte instructions. If we assume that the block size is 32 bytes, then each cache block or cache line contains 8 instructions. As of today most high performance i-caches allow us to read at least two blocks in parallel. Some caches can allow us to read even more blocks in parallel. However, two is considered a minimum, because any value less than that will severely limit the IPC of the OOO pipeline (reasons will be described in the next few paragraphs).

Now, the 4 instructions that we want to read can all be within the same cache line or they might straddle cache lines. In the latter case, we can always read two consecutive cache lines together. An example is shown in Figure 3.2.

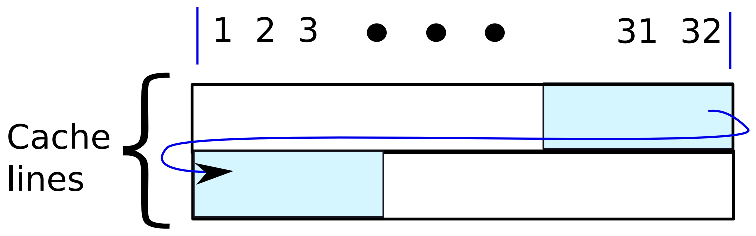

Figure 3.2: Accessing instruction bytes across two cache lines

[1]We shall use the term, PC, to refer to the program counter as well as its contents (memory address). The reader needs to figure out the right meaning (l-value or r-value) from the context.

Looking slightly deeper, we will all agree that it is necessary to be able to read at least two lines from the i-cache in parallel. Otherwise, we would not have been able to handle this case. In this case the i-cache is said to have two read ports. A *port* is defined as an external interface of an i-cache. A read port is a port through which we can only read data (and not write), and a write port is a port that allows only write accesses. Note that every memory structure needs at least one write port such that we can write data to it. Let us thus consider an i-cache with 2 read ports and 1 write port in our current discussion.

Now, let us look at the more difficult case. Assume that there is a *branch* in this set of four instructions. The probability of one among these four instructions being a branch is by no means low. Hence, we have to account for this case. If the branch is not taken then our problem is solved. A branch that is not taken is equivalent to a *nop* (instruction that does not do anything). The problem is that we do not know if a branch will be taken or not at the time of fetching the instruction. Assume that out of four instructions, the second instruction is a taken branch, and it is the only branch in this set of four instructions. In this case, the third and fourth instructions need to fetched from a different cache line. Figure 3.3 shows this situation.

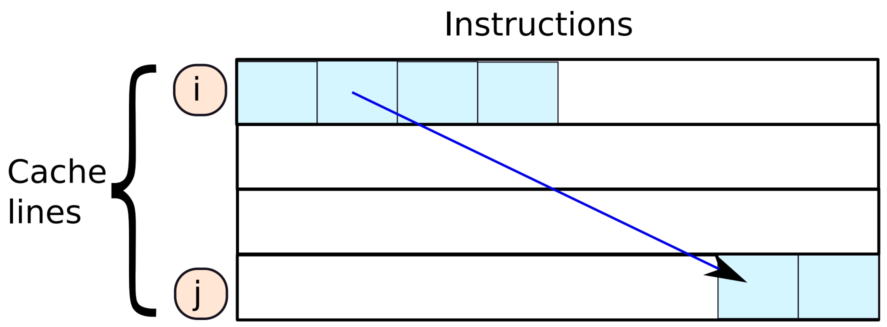

Figure 3.3: A taken branch in the middle of a 4-instruction sequence

In Figure 3.3, we see that we need to fetch the first and second instructions from cache line i, and the remaining two instructions from cache line j. Since we can fetch bytes only at the granularity of cache lines, we need to fetch line i and line j from the i-cache and pick the appropriate instructions from them.

There are some timing issues here. At any point of time, we know the address stored in the PC. Thus, we know about cache line i. There are several things that we do not know. Let us list them out:

1. We do not know which instructions in cache line i are branches. We have not had a chance to decode them.

2. We do not know if branches in line i (if there are any) are taken or not.

3. We do not know anything about the targets of the taken branches.

Given the fact that we do not know these important details before fetching the contents of cache line i, the processor would not know about line j, which contains the target instructions of a branch in line i. As a result with our current knowledge we can only design a processor that needs to work with only the instructions in line i. Since in the example we are considering, two instructions are there in line i, the maximum number of instructions that can be supplied to the remaining stages of the pipeline is two. We are thus not able to supply four instructions to the rest of the pipeline as we had originally intended to. The instruction throughput for this cycle is effectively reduced to 50% of the maximum value (4 instructions per cycle).

Important Point 3
Note that any chain is only as strong as the weakest link. In this case, we can consider a pipeline as a chain of stages where the strength of a link is determined by the number of instructions that can be sent to the subsequent stage. If the fetch stage becomes a bottleneck, as we have just seen, then the rest of the pipeline stages will remain underutilised. This needs to be avoided, and thus it is imperative to have a high-throughput fetch engine.

Now, what did we need to do to sustain a fetch throughput of four instructions per cycle? We needed to know the value of j before the current cycle. Thus, we need to design a mechanism such that the processor knows that it needs to fetch lines i and j from the i-cache. Assume that we know the value of line i from the value of the PC in the last instruction. To find out which line to fetch next (if any), we need to consider several points.

Let us outline them here.

- We need to first find if there are any branches in the instructions located at PC, PC+4, PC+8, and PC+12.
- If there are no branches, we can fetch the instructions from line i and $i+1$ (if required).
- If any of the instructions is a branch, then we need to predict its outcome and target. Assume that we predict the instruction at (PC+4) to be a branch. We further predict the branch to be taken and its target to be at address K.
- Then we need to subsequently fetch the instructions at addresses K and (K+4).

Our strategy can handle almost all the cases where one of the four instructions is a branch. However, if we have two branches, then we will not be able to fetch from three cache lines in parallel because our i-cache does not have three read ports. The likelihood of having two branches in a sequence of four instructions is low. This probability can further be reduced by instructing the compiler to avoid generating such code. The other obvious solution is to have an i-cache with more read ports.

From this discussion, here are the important problems to be solved:

Problem 1 Predict if an instruction at a given address is a branch or not without looking at its contents.

Problem 2 Predict the outcome of a branch (taken or not-taken).

Problem 3 Predict the target of a branch.

3.2 Problem 1: Is an instruction with a given PC a branch?

The question that we need to answer is as follows. We are given the PC of an instruction. We need to figure out if it is a branch instruction or not without taking a look at its contents.

Now, it is not possible to guess that an instruction is a branch without ever looking at its contents. We have to take a look at the contents of the instruction at least once, remember it, and then use this fact to make a prediction the next time we see the same PC. This is exactly what we will do. In fact what we propose next is a standard approach in computer architecture and almost all predictors work on the same principle. We shall remember the fact that an instruction is a branch (or not) when we look at its contents for the first time, and then use this information the next time we need to fetch the instruction.

Let us explain with an example. Assume that we find out that the instruction at the address 0xFFFFFFCC is a branch. We can remember this information. Just before fetching an instruction from the address

0xFFFFFFCC we can look at the information that we have collected so far, and successfully predict that the instruction at this location is a branch. This will solve our problem.

Let us now look at a standard method for remembering such kind of information. We shall create a simple predictor. Figure 3.4 shows a black box where the input is the address and the output is a prediction: is the instruction a branch or not.

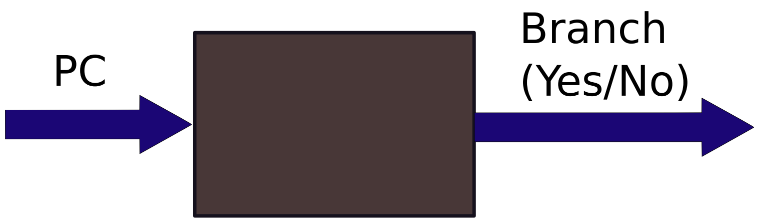

Figure 3.4: Basic structure of a predictor

A classical method for creating such a predictor is to use a table of entries, which we shall refer to as the Instruction Status Table(IST). The table is indexed by the least significant bits of the address. Each entry of the table contains the prediction: 0 (if the instruction is not a branch), and 1 (if the instruction is a branch). Let us explain with an example. Assume that the table has 1024 entries. Since $1024 = 2^{10}$, we can use the last 10 bits of the PC to locate the corresponding entry in the table.

Now, consider the fact that typically instructions are aligned to 4-byte boundaries. We can actually do better, because in this case the last two bits of the instruction's address will be 00. We can shift the address to the right by 2 places and use the shifted address to access the IST. This will unnecessarily complicate our explanation and we need to use the term shifted address everywhere. Hence, let us make a simplistic assumption in the remaining part of the text that instructions are not necessarily aligned to 4-byte boundaries. Their starting addresses can be arbitrary.

The prediction algorithm is as follows. Whenever we see an address, we extract the 10 least significant bits (LSB bits). We use these bits to index the IST and record the fact that an instruction is a branch or not. Whenever, we need to predict if an instruction is a branch, we extract the 10 LSB bits from the address, and read the IST. If the contents of the entry are equal to 1 then we predict the instruction to be a branch. This process is shown in Figure 3.5.

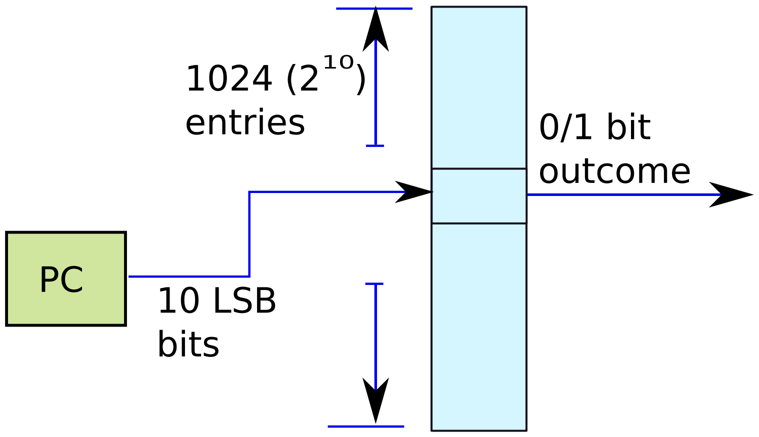

Figure 3.5: The instruction status table (IST)

Now, will this strategy work? The biggest criticism of such designs is based on the phenomenon of *destructive interference*. Let us proceed to define this term with an example. Consider addresses A and A'. Assume that both of them have their 10 LSB bits in common. They will thus map to the same entry in the table. Let the instruction at A be a branch and the instruction at A' be a non-branch ALU instruction. If we have a code sequence that alternately accesses addresses A and A', we will always make the wrong prediction. Assume that when we make a wrong prediction we replace the entry in the table with the correct prediction. When we access A, we will record the entry in the table to be a branch. Subsequently, we will try to make a prediction before accessing the instruction at A'. We will predict it to be a branch, and the prediction will be wrong. We will then set the entry in the table to 0 because A' does not point to a branch instruction. Again before accessing the instruction at A we will try to predict and get a wrong prediction (we will read a 0, should be 1). We will thus always make a mistake and the predictor will never produce the correct value. This phenomenon is referred to as destructive interference in computer architecture.

Definition 15
When two computing units or computations share a storage element, and corrupt each other's state leading to a loss in performance, we refer to this phenomenon as destructive interference. *For example, when two branch instructions share entries in a predictor table and can overwrite each other's state, we observe destructive interference. In this case this phenomenon has a name – it is known as* branch aliasing.

One thing the reader should understand is that unlike theoretical computer science, in computer architecture we do not always rely on worst case situations. We are rather optimistic people and would like to look at the sunny side of life.

Fortunately, such cases as we just described are rare. Most programs have spatial locality. This means that if we access an address, we will access nearby addresses in the same interval of time. Given that we have 10 bits, it is unlikely that we will access an address that is 2^{10} (1024) bytes away in the same interval of time. Note that for the last 10 LSB bits to be common, the difference of the two addresses with these bits in common, has to be a multiple of 1024 bytes. Secondly, both the instructions with the common LSB bits should be executed fairly frequently to have a measurable effect. This is unlikely. As a result, this simple design will not be that bad in practice; we will still get a lot of correct predictions.

Can we do better? Yes, we can. Let us consider our running example. The basic problem is that we do not maintain sufficient information. We only maintain the last 10 bits, so two addresses that have their last 10 bits in common map to the same entry in the IST. This leads to destructive interference and the instructions end up corrupting each other's state. Let us add some more information to this table to make the prediction more accurate. Let us organise it as a cache.

Recall that a cache is a memory structure that saves a subset of blocks in the memory system. Each block is typically 32-128 bytes long, and has a unique address. We can uniquely address each block in a cache, read it, and modify it. Let us organise the IST as a simple direct mapped cache. Let us assume a 32-bit addressing system, and divide the memory address into two parts: a 22-bit tag, and a 10-bit index. Let us also divide each entry of the IST into two parts: a 22-bit tag, and a status bit indicating whether the instruction is a branch or not.

The access protocol is as follows. We first divide the address into two parts and use the index to access the corresponding entry in the IST. Each entry of the IST stores 23 bits (22-bit tag + a status bit). We compare the stored tag with the tag part of the address. If they match, then we use the status bit as the prediction. This process is pictorially shown in Figure 3.6. If the tags do not match, then we predict that the instruction is not a branch (reasons explained later).

This process will eliminate the problem of using the prediction of another address. This is because we are only using the status bit if the addresses match. We will never have the case that the instruction at A will use the prediction of the instruction at A'. It is true that the least significant 10 bits of A and A' might be the same and they may map to the same entry in the IST. However, the remaining 22 bits of the addresses

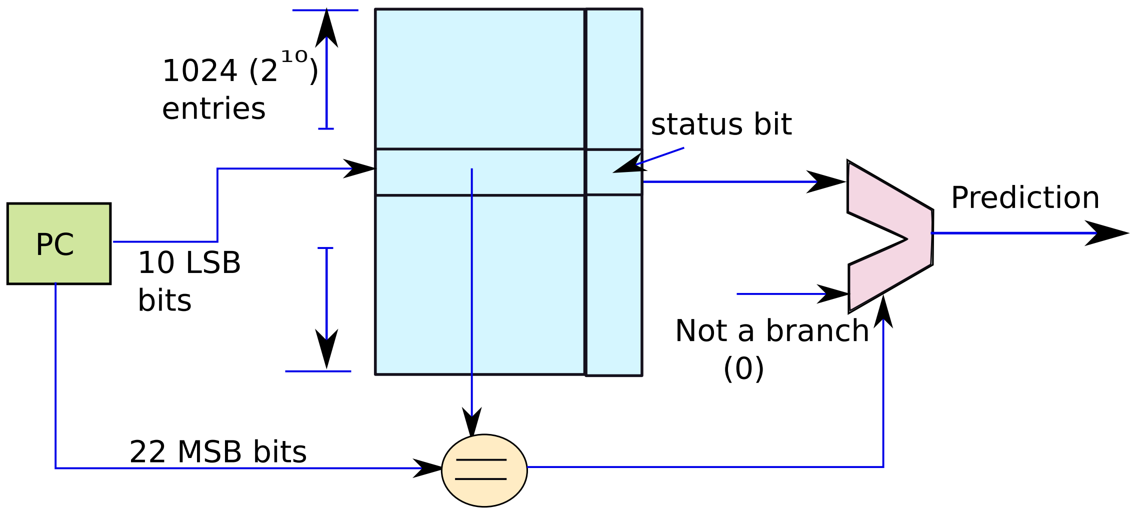

Figure 3.6: IST with tags

(the tag part) will be different. Assume that the entry for the instruction at A' is there in the IST. When we try to make a prediction for the instruction at A, the upper (more significant) 22 bits of the address will not match with the 22 bits saved in the IST's entry. As a result we will not use the status bit to make a prediction.

Let us now consider the situation in which we do not find an entry in the IST where the remaining 22 bits of the address match. In this case, it is better if we predict that the instruction is not a branch. This is because if roughly 20% of the instructions are branches, we have a 4 in 5 chance of being right.

Let us now consider the access pattern: $A \rightarrow A' \rightarrow A \rightarrow \ldots$. With our simple IST the prediction rate was 0. With this version of the IST that is organised as a direct mapped cache, our hit rate will be better. The reasons are as follows. The instructions at A and A' will be overwriting each other's entries. When we access address A, we will not find its entry in the IST; we will instead find the entry for A'. The case for A' will be likewise. Since we will never find an entry, we will always predict that the instruction is not a branch. We will be right 50% of the time.

We have thus magically increased our hit rate from 0% to 50%!

This was a somewhat contrived example. The readers are invited to conduct simulations using real world benchmarks, implement both the predictors, and take a look at the results.

Activity 1
Get a trace of branch instructions using an architectural simulator such as the Tejas Simulator^{TM} [Sarangi et al., 2015], and do the following.

1. *Get a trace of the program counters for branch instructions. The trace should have at least 1 million instructions.*
2. *Implement different versions of the IST and measure the accuracy of predicting whether a given PC corresponds to a branch instruction or not.*
3. *Vary the number of tag bits from 0 to 22.*

3.2.1 Recording the Type of the Branch

We broadly have three kinds of branches: conditional, unconditional, and function calls/returns. They need to be handled differently. For conditional branches, we need to predict the outcome of the branch. For unconditional branches, the outcome is already known. All such unconditional branches are taken branches. Technically, function calls and returns are also unconditional branches because they are always taken. However, we choose to treat them as a separate category because we can create a really efficient mechanism to handle them (see Section 3.4.2).

In this section we are concerned with the mechanism that records and predicts the type of an instruction. We can implement this very easily by creating more status bits. Let us have 3 status bits, where the combination of bits indicates the type of the instruction. Refer to the following table.

Status Bits	Instruction type
000	Not a branch
001	Conditional branch instruction
010	Unconditional branch instruction
011	Function call
100	Function return

By having 3 status bits in each entry of the IST, we can correctly record the type of the instruction, and use it for further processing. We need the outcome of the branch, only if the instruction is a conditional branch. For unconditional branches, function calls and returns, we just need the target of the branch. We shall discuss methods to predict the branch target in Section 3.4.

3.3 Problem 2: Is a branch taken or not taken?

Now, that we know with some certainty that an instruction is a branch, we need to predict its direction (taken or not taken).

Let us first set the baseline. Assume that a branch's outcome is genuinely random akin to the outcome of a coin toss. It is possible to prove that irrespective of our approach we will never be able to successfully predict more than 50% of the outcomes. If we just simply guess that the branch is taken all the time, we will predict it successfully with a probability of 0.5. This statement can be theoretically proven using the Fano's inequality [Cover and Thomas, 2013] in information theory. This is beyond the scope of this book. Interested readers can always consult a book on information theory.

Let us now start our discussion by considering one of the simplest branch predictors namely the bimodal predictor.

3.3.1 Bimodal Predictor

We should shun the pessimistic point of view that states that branches are totally random. The reason is that branch outcomes are not totally random and thus it is possible to design much better predictors, whose accuracy is more than 90-95%. Let us consider a few examples. Consider the following C code snippet.

```
i = 0;
while (i < 100) {
        i = i + 1;
        ...
}
```

Here, the branch corresponding to the *while* loop is not taken (control remains within the body of the *while* loop) for 100 times, and then it is taken (exits the *while* loop) just once. In this case the behaviour of the branch instruction is predictable. Let us assume that when we enter the body of the *while* loop the branch is predicted to be not taken. Thus, if we predict the branch to be not taken, we will be right 100 out of 101 times. Our accuracy will be close to 99%.

Let us now consider the following code snippet.

```
i = 0;
while (i < 100) {
    i = i + 1;
    if (i%10 == 0) {
        printf ("Multiple of 10");
    } else {
        printf ("Not a multiple of 10");
    }
}
```

We need a branch corresponding to the *if* statement. Assume that we enter the *else* part if the branch is taken, otherwise if the branch is not taken we enter the body of the *if* statement. We can clearly see in this example that 9 out of 10 times this branch will evaluate to taken.

In both these examples we have seen branches that most of the time evaluate to either taken or not-taken. It should be possible to design a predictor to predict them based on their activity in the past. We always do this in our lives. If a given store has historically been offering a discount, we choose it for shopping as compared to similar stores that do not offer discounts. There is no guarantee that on any given day the store will offer a discount. We can just make an intelligent guess based on its past behaviour.

Let us do something similar here. For each branch, let us save its history in a table. The next time that we encounter the branch instruction, we can read its history from the table and make a prediction. The history needs to be updated each time we become aware of the actual outcome of the branch.

Let us propose a simple design that is similar to the IST. Note that branch predictors need to be very simple structures because they lie on the critical path. Furthermore, we need to start predicting the outcome of a branch immediately after its PC is known and we need to finish the prediction before we proceed to fetch the instruction that can be the branch target. To avoid any pipeline stalls we need to ensure that the branch prediction finishes within one processor cycle.

Let us have a simple table similar to the IST with 2^n entries, which is indexed with the last n bits of the PC as shown in Figure 3.7. We can store the latest outcome of a branch: taken or not-taken. We thus need to store a single bit: 0 for not-taken and 1 for taken. We use this bit as the prediction (for the future). This means that if last time a given branch was taken, we predict it to be taken the next time. However, this is not a good approach.

Consider a function foo that contains a *for* loop with 5 iterations (refer to Figure 3.8). The branch associated with the *for* loop (*beq .exit*) will evaluate to not-taken 5 times, and the control will enter the body of the loop. The 6^{th} time the branch will be taken and we will not enter the body of the loop. If we save the last outcome of the branch in each entry of the predictor table and use it to predict the outcome of the branch the next time we encounter it, we will make a misprediction the 6^{th} time, because in the 5^{th} iteration we would have recorded the fact that the branch was not taken. We will thus predict that the 6^{th} time also, the branch will not be taken. This is wrong.

Now assume that we call the function foo that contains the for loop once again? We will mispredict the branch associated with the first iteration of the for loop because the last time we had encountered the branch, it was taken (not entering the for loop), and we would use this fact to make a prediction.

Let us now assume that we call the function foo repeatedly and there is no branch aliasing. The branch associated with the for loop will be invoked 6 times (5 times for entering the loop, and once for not entering it) for each function invocation. We will mispredict the branch twice (1^{st} and 6^{th} iterations). Thus, the misprediction rate is $2/6 = 1/3$, which is significant. Let us try to reduce it to $1/6$.

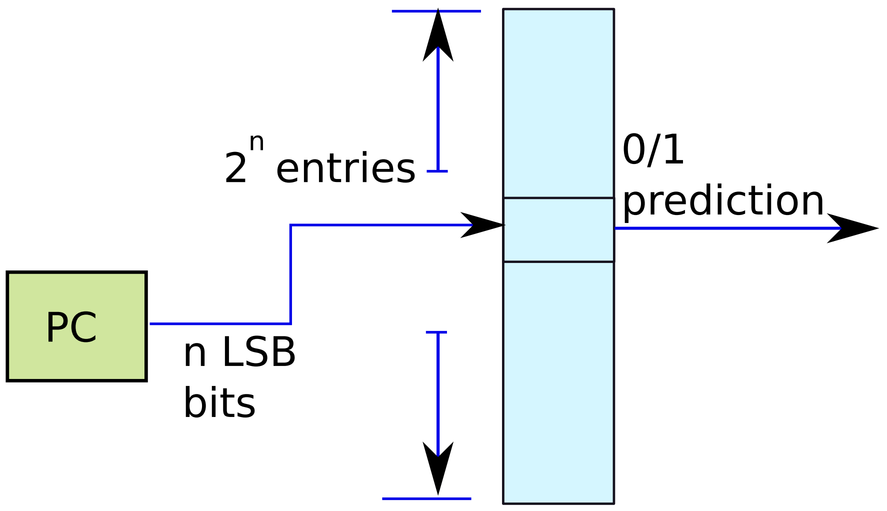

Figure 3.7: A simple single bit branch predictor

C code

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

Assembly code

```
    /* i mapped to r1 */
        mov r1, 0
.loop:
        cmp r1, 5
        beq .exit
        ...
        add r1, r1, 1
        b .loop
.exit:
        ret
```

Figure 3.8: A loop with 5 iterations

3.3.2 Predictor with Saturating Counters

The conditional branch in the *for* loop evaluates to not-taken 5 out of 6 times. The last invocation when the branch is taken can be thought of as an exception. The next time that we call the function *foo*, we should predict the branch (corresponding to the first iteration) to be not-taken based on its past behaviour. We should have a mechanism of either discarding or giving less weightage to exceptions.

Instead of keeping 1 bit to predict the outcome, let us instead use 2 bits in each entry of the predictor table. Let us refer to these 2 bits as a counter, where the count is the binary number represented by 2 bits. If the branch is taken, we increment the counter, and likewise if the branch is not taken, we decrement the counter. Since we have 2 bits, they can only be in the range of 00 to 11. If the count is equal to 11, and we find that the branch associated with it is taken, let us not further increment it. Let us instead maintain it at 11. Similarly, if the count is 00, and the branch is not taken, let us keep the count at 00. Such kind of a counter is known as a *saturating counter*.

Definition 16
A saturating counter is a special kind of counter that has the following fields: the count (C), a lower threshold (L), and an upper threshold (U). It supports two operations, increment *and* decrement*, which are defined as follows:*

increment: $C \leftarrow min(C+1, U)$

decrement: $C \leftarrow max(C-1, L)$

The counter thus cannot count beyond L on the lower side, and U on the higher side.

Let the prediction table have 2^n 2-bit saturating counters. We increment or decrement the counter once the outcome of a branch is known. Let us now use this array of saturating counters to predict the outcome of a branch.

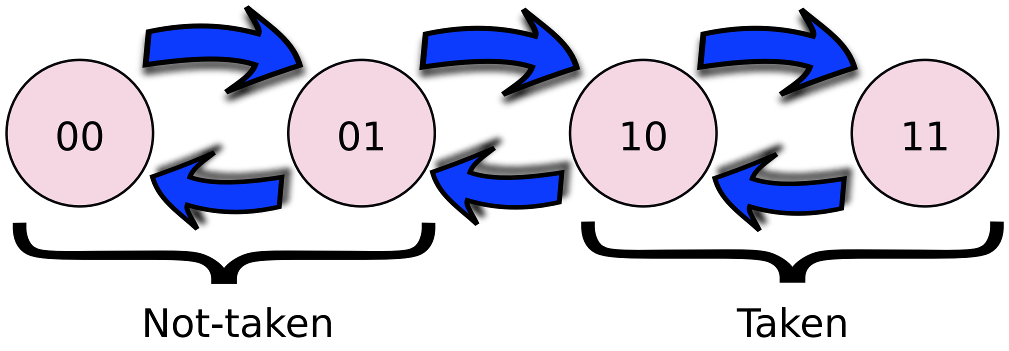

Figure 3.9: The state diagram of a saturating counter

Let us use the states 00 and 01 to predict not-taken, and the states 10 and 11 to predict taken. Figure 3.9 shows the state transition diagram of a saturating counter. If a branch is found to be taken, we move from left to right and keep on incrementing the counter. Once we reach the end (state 11), we do not further increment the counter. Similarly, if a branch is found to be not taken, we move to the left till we reach state 00. Let us use this array of predictors with our running example (function foo with a for loop in Figure 3.8).

Let us assume that we start in the state 01 (predict not-taken). After the first 5 iterations, the state of the counter will be 00. We will mispredict the branch in the 6^{th} iteration, and the state of the counter will be set to 01 since it is a taken branch. Hence, our misprediction rate for this invocation of function foo is 1/6.

Let us see what happens when we call the function foo once again. We start with the state 01. For the first iteration of the for loop, we predict it to be not taken (value of the counter = 01), which is correct. Subsequently, the count remains at 00 till the 6^{th} iteration when it again becomes 01. Here also the misprediction rate for the branch in the for loop is 1/6.

Note that if we would have started with a different value of the counter, the final value would have still been 01 because we encounter five consecutive not-taken branches. Irrespective of the starting value, the value of the counter at the end of the 5^{th} iteration of the for loop will be 00. Subsequently, we encounter a taken branch, and the counter gets set to 01. If we call the function foo a large number of times the misprediction rate for the conditional branch in the for loop will stabilise to 1/6.

We can make an important conclusion here. By switching from the predictor with 1-bit counters to a predictor with 2-bit saturating counters we have halved the misprediction rate.

The states 00, 01, 10, and 11 also have the following alternative names.

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

Given our discussion, we are in a position to understand why these states have been named in this manner. Let us assume that the counter is in state 00. We can tolerate one misprediction and still keep on predicting not-taken as we have done with our running example. Hence, this state is known as "strongly not-taken". Likewise is the case for state 11. The states 01 and 10 do not have that strong a bias. A single misprediction can change the subsequent prediction. For example, if the state is 01, and we have a misprediction (branch is taken), the new state is 10. We now predict the branch to be taken. Thus, this state is known as "weakly not-taken".

By replacing single bits with saturating counters, we have made the predictor slightly more intelligent. It can take a longer history of the branch into account, and it can effectively filter out the effects of infrequent outcomes. We can always extend this argument to have 3 or 4-bit saturating counters. However, there are negative aspects of increasing the number of bits. The first is the size of the predictor, and the associated area and latency overheads. The other is that the predictor becomes less responsive to changes in the branch outcomes. Assume that a branch for a long time was not being taken. The count will stabilise to 0. After that if suddenly we start taking the branch, a 2-bit counter will reach the weakly taken and strongly taken states sooner than a 3-bit counter. For a 3 or 4 bit counter, we will have many more intermediate states that will predict the branch to be not-taken. Computer architects typically run very extensive simulations with benchmarks to find the best possible configuration for a branch predictor.

3.3.3 Loop Predictor

Let us now look at designing a better predictor. The main problem with our running example is that we were not able to predict the last branch in the loop with a simple saturating counter based predictor. This is because the saturating counter associated with the branch gets set to the strongly not-taken state by the time we reach the end of the 5^{th} iteration.

Let us first discuss a solution with limited applicability. Assume that we know the number of iterations at compile time. For example, we know that a loop will execute for only 100 times. We can create a simple circuit called a *loop predictor* that contains a register, and a small adder. When we enter the loop for the first time, the compiler can add an instruction to set the count in the register to 100. For every iteration of the loop we can decrement the count in the register. This can either be done via a dedicated instruction added to the program or the hardware can automatically figure out a loop iteration by looking at the contents of the instruction. Most loops are implemented using *backward branches*. These branches have a negative offset with respect to the program counter, which means that the control jumps to a point that is before the current instruction in the program. Most of the time, programs with loops use such branches to jump to the beginning of a loop. It is also possible for the compiler to add additional flags to an instruction to indicate that it is at the start of a loop iteration.

Now, if an instruction's PC is the same as that of the PC of the branch that decides if we need to iterate a loop, we can be sure that we are either entering a loop's iteration or exiting the loop. Once, the count reaches 0 we can predict that we need to exit the loop. In this case, when the count reaches 0, we can predict the branch (associated with the loop) to be taken and thus there will be no mispredictions. This solution undoubtedly sounds simple and effective. However, it is not generic. Most of the time, the number of iterations of a loop is not known at compile time. We can have *break* statements in a loop, loops inside a loop (nested loops), and loops might call complex functions with loops inside them. We can always make our simple design more complicated by having an array of registers indexed by the last few bits of the PC to accommodate multiple loops. Architects need to carefully evaluate the trade-offs while designing and using such predictors.

A general criticism of any approach that requires additional support from the compiler is that the approach does not remain generic anymore. A program that has special instructions might not run on a processor that does not support those instructions, but supports most of the other instructions in the ISA.

3.3.4 Predictors with Global History

We have looked at the local history of a branch, which is its behaviour in the past. Let us now look at the global history, which is the history of the last few branches of the program: they can either have the same PC or they can have different PCs.

Assume the code has an *if statement* before the end of the loop that does some further processing. It checks if the loop variable i is equal to 4. Assume that there are no other statements with branches inside the body of the *for* loop.

C code

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

Assembly code

```
/* i mapped to r1 */
mov r1, 0
.loop:
    cmp r1, 5
    beq .exit
    ...
    cmp r1, 4
    bne .cont
    /* Inside the body of the if
       statement */
    ...
.cont:
        add r1, r1, 1
        b .loop
.exit:
        ret
```

Clearly, the *if* statement's outcome is related to the outcome of the branch instruction in the *for* loop when we are exiting the loop (i is equal to 5). When i is equal to 4, we will enter the body of the *if* statement. We shall subsequently set i to 5, and conclude that we need to exit the loop. Let us create a circuit to capture this pattern.

We shall call this circuit the *branch history register* (BHR). This n-bit register records the outcome of the last n branches in a bit vector. For example, let us consider a 3-bit register. Let us refer to a taken branch by the bit 1, and a not-taken branch by the bit 0. If the last three branches are taken, the contents of the BHR will be 111. If the last branch was taken, and the branches before it were not taken, the contents of the vector will be 100. Whenever, we know the outcome of the branch we shift the contents of the BHR to the right by 1 position, and set the outcome of the latest branch as the MSB (most significant bit).

Let us now consider our *for* loop. Assume that we enter the *for* loop if the branch associated with it is not taken. Similarly, let us assume that we enter the body of the *if* statement if the branch associated with it is not taken. Let us assume a BHR with 3 bits. This means that the BHR contains the state of the last three branches encountered by the program. Note that the *bne* instruction refers to branch if not equal.

Let us compute the state of the BHR at the end of the 5^{th} iteration. The last three branches and their outcomes are as follows:

Branch	Outcome
if statement in the 5th iteration	0 (not taken)
for loop branch at the beginning of the 5th iteration	0 (not taken)
if statement in the 4th iteration	1 (taken)

Thus the contents of the BHR are 001. This combination will only occur in the 5th iteration only. Thus, we can use the contents of the BHR to predict that we are done with the enclosing *for* loop and we can exit it. When we come to the branch that decides whether we need to enter the body of the *for* loop or exit it, we can use the contents of the BHR and safely predict that the branch should be taken, or in other words, we should not enter the *for* loop if the contents of the BHR are 001. This is known as the *global history*, because this information refers to the behaviour of other branches. In comparison, when we record the history of only a given branch instruction, this is known as the *local history*.

Definition 17

- *The global history captures the behaviour of the last* k *branches that have been encountered in program order.*
- *The local history captures the behaviour of only a single branch.*

Let us now design a predictor that uses this information to make a prediction.

3.3.5 A 2-Level Predictor

Such a predictor uses a BHR. Its design is shown in Figure 3.10.

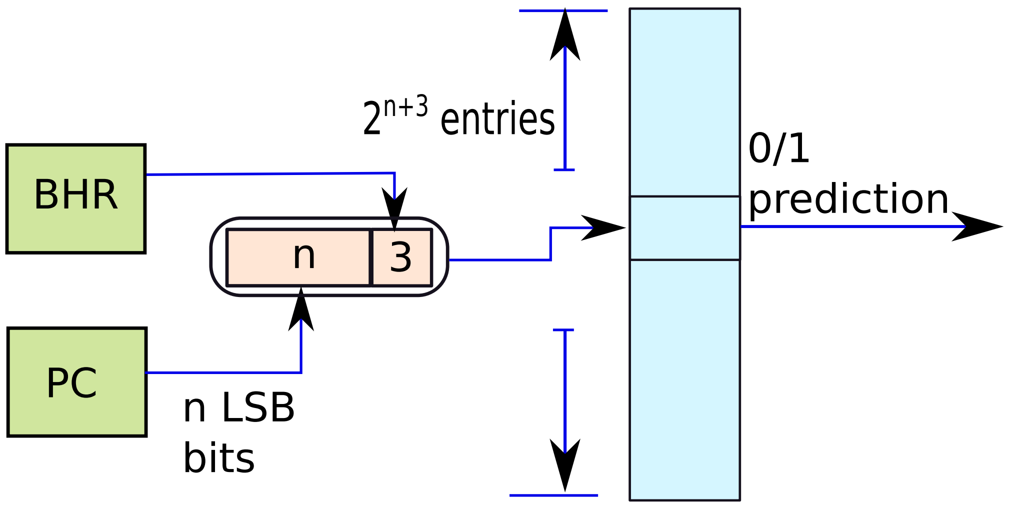

Figure 3.10: A global predictor that uses the BHR

Here, we combine two sources of information. We consider the contents of the 3-bit BHR (branch history), and the last n bits of the PC. Together, we have $n + 3$ bits. We use these $n + 3$ bits to index [2] a table with

[2]Indexing a table in computer architecture means accessing an entry in the table with the given address.

2^{n+3} 2-bit saturating counters. The prediction and training steps are similar to that of the bimodal predictor with saturating counters. The only difference is that we are considering a tuple of the branch history and the PC bits to access the table.

Let us now argue that this is a better predictor than the bimodal predictor by considering our running example (*for* loop with 5 iterations). Let us consider the steady state where the code containing the *for* loop has been invoked many times. Now, consider the branch associated with the *for* loop. Only if the global history is equal to 001 (see Section 3.3.4), the branch should evaluate to taken. Otherwise, for the 7 other values of the global history the branch should evaluate to not-taken. Given that for each PC (represented by its last n bits), we have 8 entries (one for each combination of the bits in the BHR), we can store different outcomes for each value of the BHR. We store taken only for the combination 001. For the rest of the combinations we store not-taken.

Now, when we enter the first iteration of the *for* loop, it is highly unlikely that the BHR will contain 001. The likelihood of some other combination of bits being present in the BHR is high, and thus we will most likely predict not-taken. As we proceed till the last iteration, our predictions for the *for* loop will remain as not-taken. However, when we predict the outcome of the *for* loop's branch for the 6^{th} time, the contents of the BHR will be 001, and we will predict taken (based on past behaviour), which is correct. After we exit the loop, the state of the corresponding BHR will be 100. The next time we enter the *for* loop, if the BHR remains the same (if there is no aliasing), we will make the correct prediction: not-taken. As a result, our current predictor will not make any mistakes. The rate of mistakes for the branch associated with the *for* loop will be 0.

Such kind of predictors are known as 2-level predictors. The first level reads the global history, the second level uses the global history, and bits of the PC to index a table of saturating counters. The table of saturating counters is also known as the *pattern history table* (PHT). For each pattern of bits (from the PC and BHR) it stores a saturating counter.

3.3.6 GAg, GAp, PAg, and PAp Predictors

Now, that we have seen the 2-level predictor, let us explore the space of all kinds of predictors that use a combination of bits from the PC and the BHR (branch history register).

Let us elaborate. In Section 3.3.5, the first level uses the BHR to record the behaviour of the last m ($m = 3$) branches. The second level predictor uses the contents of the BHR and n bits from the PC to create an $n + m$ bit number. This number is used to index a table of saturating counters. We argued that this is a good idea because it makes use of the behaviour of the last few branches. It is possible that the behaviour of the last m branches might give a clue about the behaviour of the next branch. This was indeed the case as we saw in the example code provided in Section 3.3.4.

Let us now classify each level into two categories: global(G), and local or per-address(P). A level is global if it uses structures that do not require bits from the PC to access it. For example, the BHR in the 2-level predictor that we saw in Section 3.3.5 is an example of a global structure. It is not indexed by bits from the PC – it is global in that sense. However, in the same predictor the next level uses bits from the PC. It stores some information that is specific to a set of instruction addresses. This is why the next level can be classified as *per-address*. Hence, the predictor described in Section 3.3.5 is a GAp predictor: global at the level of the BHR and local at the level of the PHT (table of 2-bit saturating counters).

On the same lines, we can have three more types of predictors – GAg, PAg, and PAp – as described by Yeh and Patt [Yeh and Patt, 1992]. Let us quickly look at them.

GAg Predictor

This is a very simple predictor that does not use any bits from the PC. The first level is a BHR (similar to our 2-level predictor described in Section 3.3.5), and the second level is a table that is indexed by the bits stored in the BHR. This is shown in Figure 3.11. We assume that the BHR stores the behaviour of the last m branches. We use the contents of the BHR (m bits) to index the PHT (table of 2-bit saturating counters).

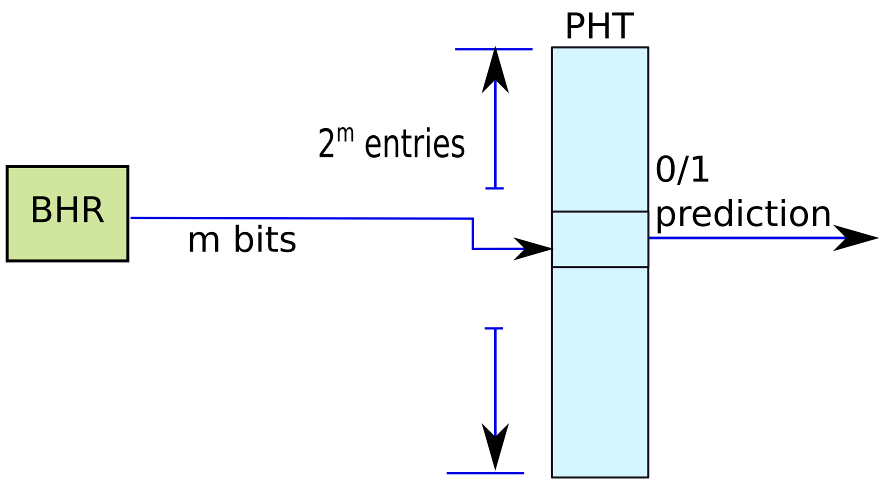

Figure 3.11: A GAg predictor

The advantage of this design is simplicity. However, the disadvantages are plenty. We are completely ignoring the past behaviour of any given branch. Assume it is always taken. We have no way of recording this information. We are completely basing our outcome on the behaviour of the m preceding branches, which is not the best thing to do in this case. This is because there can be many scenarios in a large program where the patterns for the last m branches will be identical. The associated outcomes might be very different though. This will cause an erroneous prediction for branches that are highly predictable (mostly taken or mostly not taken), and cases where the branch outcome is highly correlated with the past behaviour of the same branch.

The GAg predictor is thus not used in practice.

PAg Predictor

The first level in this case needs to use bits from the PC. Let us assume that we use 3 bits from the PC. There are thus 8 possible combinations. Each combination is associated with a set of program counters (PCs). Instead of one BHR, let us have eight BHRs – one for each set of PCs. This is shown in Figure 3.12. We show that we extract 3 bits from the PC address, and then access the appropriate BHR. Let us read an m-bit pattern P from the corresponding BHR.

We use this pattern P to access the pattern history table (PHT). Since this is a PAg predictor, we have a single table in the second level. Treating P as a binary number, we access the P^{th} entry in the PHT.

Note that as compared to the GAp predictor (Section 3.3.5), we do not use any bits from the PC to access the PHT. Let us understand the pros and cons of this design.

The PAg and GAg predictors have the same disadvantage. They do not take the past history of any given branch instruction into account. Considering the behaviour of other branches is not an effective substitute of the per-branch history. The same example as cited in the case of a GAg predictor holds here also. Assume a highly predictable branch (always taken or not taken), we might often be lead to wrong conclusions if we just consider the behaviour of other branches. This information is not specific, and a lot of aliasing is possible.

However, the PAg predictor has some advantages as well. Let us see why. We use multiple BHRs, and this is often a good thing. To understand the reason behind this, let us look at the first level. We pick a couple of bits from the PC, and use them to choose the appropriate BHR. Which bits should they be? Should they be the least significant bits? The answer is probably no. Let us explain with an example. Assume we take 3 ($k = 3$ in Figure 3.12) bits from the PC. If they are the least significant bits then the 1^{st}, 9^{th}, and 17^{th} branches in the program will map to the same BHR. There is little in common between them, and it is

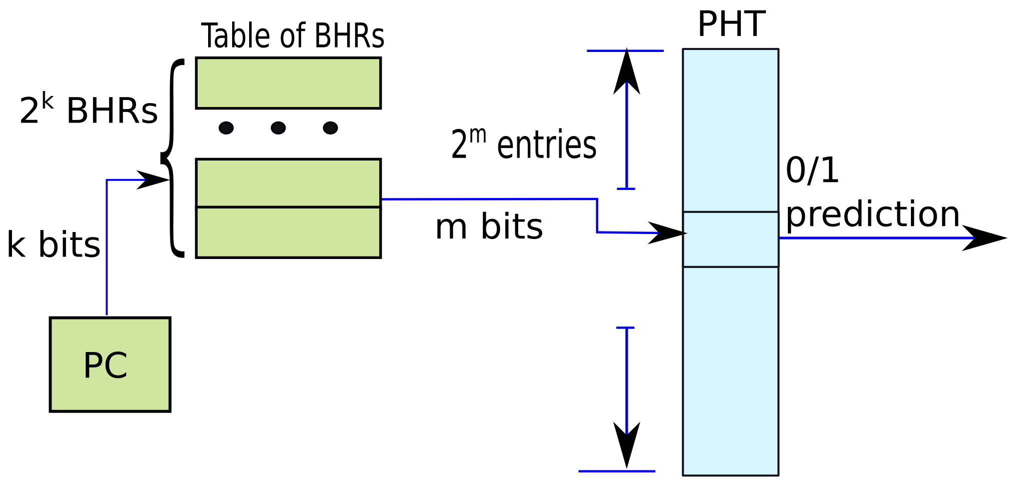

Figure 3.12: A PAg predictor

unlikely that we will get any benefits. Even worse, we will have the problem of aliasing, and the results will degrade.

Now, if we take the 3 most significant bits of the address, then all the branches in large regions of code will map to the same BHR. Again this is a bad idea, because effectively we have just one BHR. We are not reaping the advantages of having multiple BHRs. What is the solution?

The solution is to take 3 bits from somewhere in the middle of the address. Assume that these are bits 11, 12, and 13. In this case, all the branches in a 1 KB [3] region of code map to the same BHR. This is a good thing because this is in line with our original aim of having a BHR, which was to record the behaviour of recent branches. This is happening here. Let us now explain the logic of having different BHRs for different sets of addresses.

Because of spatial locality, in any window of time we are not expected to be touching a very large region of code. We will mostly be limited to small regions that contain loops. Assume that we are limited to a region of 5 KB. In our current mapping scheme, where each block of 1 KB is mapped to a separate BHR, we will map this 5 KB region to 5 separate BHRs. We will thus not have any aliasing and the consequent destructive interference. Furthermore, for every branch its corresponding BHR will contain the behaviour of the last n branches that are close to it in terms of the PC address (are in its neighbourhood). Branches that are far away and unrelated, will not be able to corrupt the state of the BHR. The BHR will thus capture more accurate information.

Let us try to summarise this argument. The aim of a BHR is to capture the behaviour of branches within a given neighbourhood of addresses. If we have a single BHR, then the possibility of destructive interference by branches in other regions of the program arises. Hence, it is the best to give a different BHR to each distinct neighbourhood of PC addresses. This is something that this predictor is trying to achieve.

When we move to a very different region of code, then the new branches will overwrite the contents of the BHRs. However, this is fine, because we expect to stay within the new region for a considerable duration of time given that most programs have a lot of temporal locality. The BHRs will get repopulated, and will capture the branch patterns in the new region.

[3] If we start the count from 1 (for the LSB), then each combination of the 11^{th}, 12^{th}, and 13^{th} bits points to a region of code that takes 10 bits to address. The size of this region is thus 1024 bytes or 1 KB.

PAp Predictor

This is the most generic predictor. It uses bits from the PC to access tables in both the levels. The first level looks like the first level of the PAg predictor (BHR selected by k PC bits). The second level looks like the second level of the GAp predictor: m bits from the relevant BHR concatenated with n bits from the PC. We use a combination of the pattern read from the BHR, and a few PC bits to access the second level table of saturating counters (the PHT). Refer to Figure 3.13.

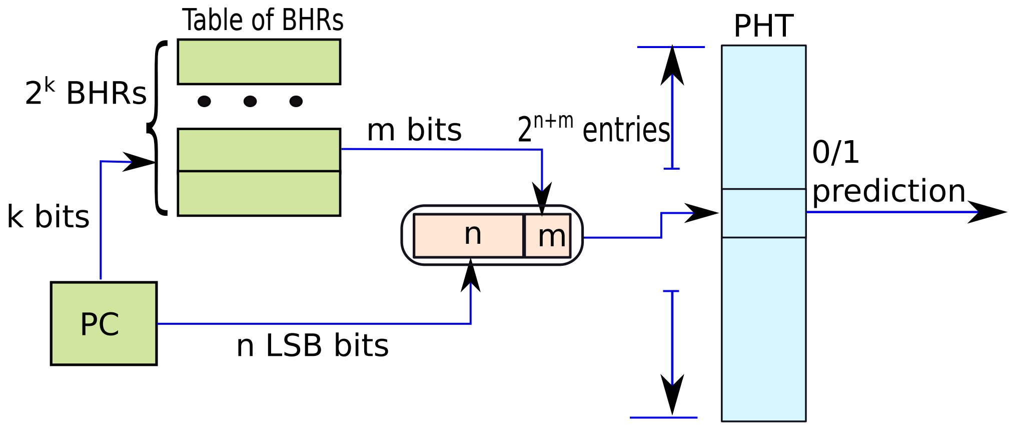

Figure 3.13: A PAp predictor

This scheme does not suffer from the disadvantages of the GAg and PAg predictors. It in fact, enjoys the advantages of both the GAp and PAg predictors. At each level, we need to very judiciously choose the the number and positions of bits that we want to take from the PC, and we also need to carefully choose the number of entries in the BHRs. If we are able to make a good choice, then such predictors typically have very high prediction rates.

A word of caution is due. Even though the PAp predictor has advantages, it need not always be the best choice. The first problem with it is that it is a large predictor – much larger than most of the other predictors that we have described. Large predictors are slow, are power consuming, and require a lot of area.

Additionally, it takes a lot of time to train such predictors. For example, a single-bit bimodal predictor can be trained very easily. If there are 1024 entries in the predictor, then all it takes is 1024 unique accesses to train it completely. If we have 2-bit saturating counters, then we require at the most 3 accesses to each entry to train it (e.g: strongly taken $\rightarrow$ strongly not taken). Now, if we have a BHR, we need to train the predictor for each access pattern in the BHR, which will require even more accesses. If our code size is small, or if we have a very high degree of locality where we are guaranteed to remain within the same region of code for millions of cycles, then the PAp predictor makes sense. However, for codes with low temporal locality, we will not be able to properly train the PAp predictor. By the time, we have trained the PAp predictor, we would have already moved to a different region of code, and all our previous work will get wasted.

3.3.7 GShare Predictor

We saw that the PAp predictor is the most powerful, yet suffers from various disadvantages. The primary disadvantage is the large size and the large training time. Even if we are not bothered about the training time, the on-chip area and power consumption are significant issues. Let us try to do the same thing in a different way.

The main aim while designing our predictors was to somehow combine global information stored in the BHRs with local (per branch) information. Let us extend the GAp predictor in a novel manner.

Let us have a single BHR in the first level. This will record the behaviour of the last n branches in a bit vector. In the GAp predictor, we were concatenating the pattern read from the BHR and m bits from the PC. In this case, let us instead do something else. Let us compute the XOR (exclusive OR) of the bits. The idea is to take the m-bit pattern from the BHR and perform a XOR operation between it and m PC bits.

In the second level, we access the PHT using the result of the XOR operation as an index (see Figure 3.14).

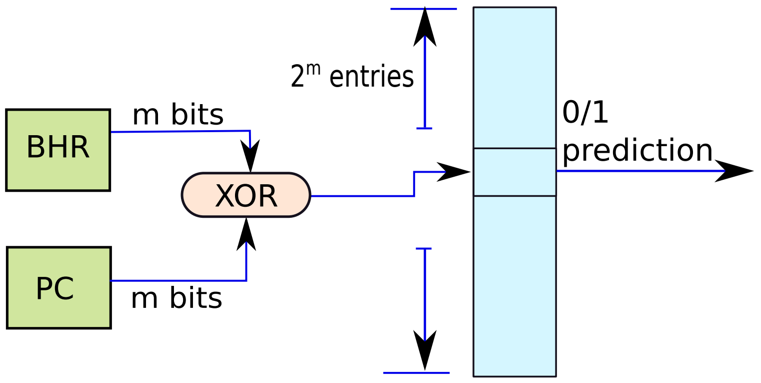

Figure 3.14: The GShare predictor

Using the XOR function as a method to combine two sources of information – PC bits and the global branch history – is an innovative idea. We can also choose n PC bits, and m BHR bits, and XOR them. When $n \neq m$, we need to pad the number with fewer bits with zeros while doing the XOR operation. The idea here is that after computing a XOR we will most likely arrive at a unique combination of PC bits and BHR bits. There is still a likelihood of aliasing. However, if n and m are large enough, and we have temporal and spatial locality in the code, then most likely in a small window of time the output of the XOR operation will uniquely represent the pair of bit vectors (PC bits and the branch history).

Recall that in the GAp and PAp designs, we were concatenating the PC bits with the branch history to get a unique combination; in this case, we are achieving something roughly similar by using a lesser number of bits. The trick is to use the XOR operation to combine two disparate sources of information.

Given that the probability of aliasing is relatively low in practical scenarios (as we have argued), using the GShare predictor is expected to be beneficial for performance. Furthermore, the number of rows in the PHT is $2^{max(m,n)}$ as compared to 2^{m+n} in the GAp and PAp predictors. This means that the table of saturating counters is smaller, faster, and more power efficient. This makes GShare a fast and efficient predictor.

3.3.8 Tournament Predictor

We have discussed a lot of predictors starting from the simple bimodal predictor to the GShare predictor. As we have argued, no predictor is good for all cases. For example, a PAg predictor is good when per-branch history does not matter and the behaviour of a branch is mostly dependent on the behaviour of the last n branches. As compared to that, a table of saturating counters is the best when branches are simply not dependent on other branches.

The accuracy of a predictor depends on the characteristics of the underlying code that the machine is executing. It is possible that one predictor might be good for one kind of code and another predictor might be the most suitable for another kind of code. There should be a mechanism of choosing one predictor for one region of the code, and another predictor for a different region.

Let us thus create a new kind of a predictor called a tournament predictor that contains multiple predictors. Refer to Figures 3.15 and 3.16. Here, we have two predictors: $Pred_1$ and $Pred_2$ (Predictor 1 and Predictor 2 in the figures). We then have an array of saturating counters called the selector array. It is a table of 2-bit saturating counters and is indexed by n bits of the PC. It is used to choose the outcome of one of the predictors.

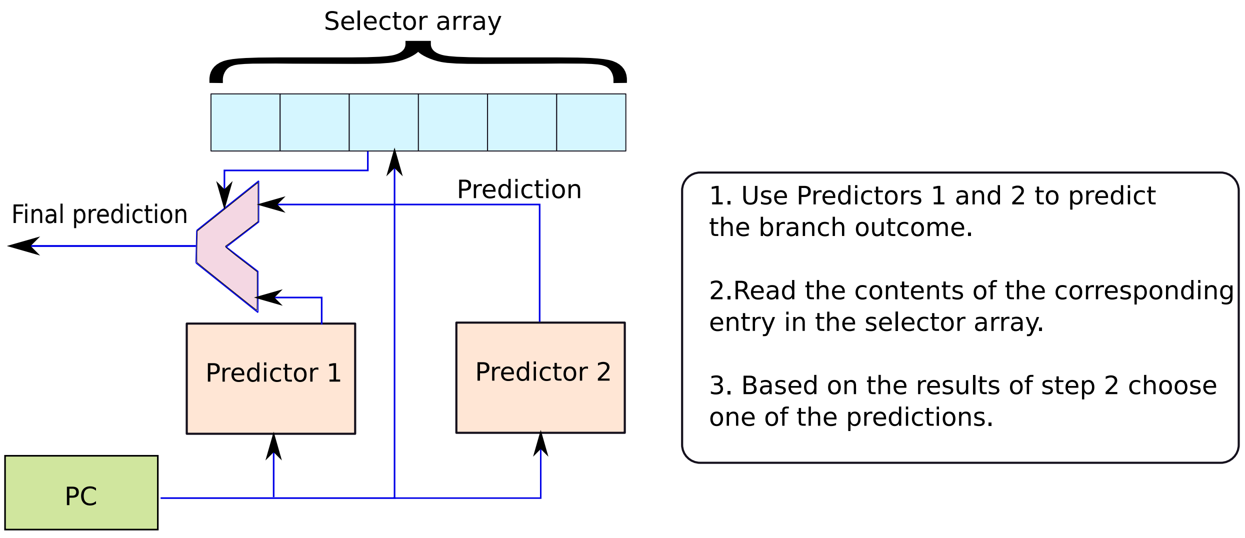
Selector array
Prediction
Final prediction
Predictor 1
Predictor 2
PC
1. Use Predictors 1 and 2 to predict the branch outcome.
2.Read the contents of the corresponding entry in the selector array.
3. Based on the results of step 2 choose one of the predictions.

Figure 3.15: Branch prediction using the tournament predictor

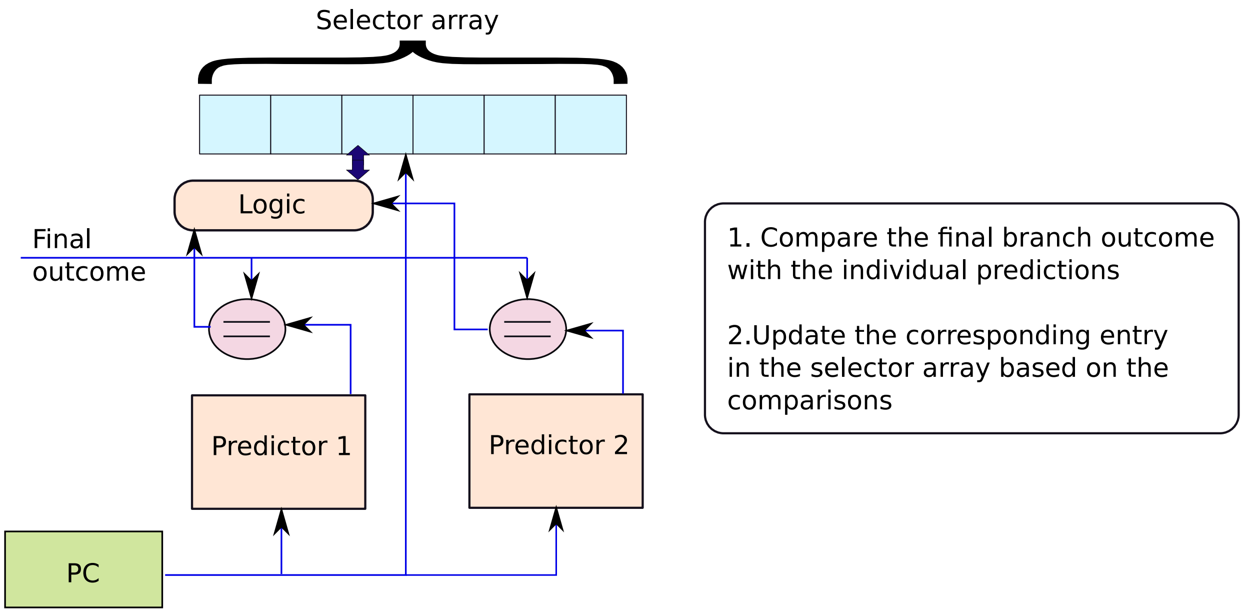
Selector array
Logic
Final outcome
Predictor 1
Predictor 2
PC
1. Compare the final branch outcome with the individual predictions
2.Update the corresponding entry in the selector array based on the comparisons

Figure 3.16: Training the tournament predictor

The operation of the predictor is as follows. We first access the corresponding entry in the selector array for a given PC. If the value of the saturating counter is 00 or 01, we choose $Pred_1$, otherwise we choose $Pred_2$. Then we access the chosen predictor ($Pred_1$ or $Pred_2$), and use it to predict the branch. In parallel, we also run the other predictor (the one not chosen) and record its output.

Once the outcome of the branch (referred to as the Boolean value *result*) is known, we need to train the predictors. First, we train both the predictors separately using the outcome of the branch. The predictors internally update their tables. Now, we need to update the selector array. The logic is as follows. Assume that the function $outcome(Pred_1)$ refers to the outcome of $Pred_1$, and $\wedge$ represents a logical AND.

- $outcome(Pred_1) = outcome(Pred_2) \rightarrow$ Do not do anything.
- $(result = outcome(Pred_1)) \wedge (outcome(Pred_1) \neq outcome(Pred_2)) \rightarrow$ Decrement the saturating counter.
- $(result = outcome(Pred_2)) \wedge (outcome(Pred_1) \neq outcome(Pred_2)) \rightarrow$ Increment the saturating counter.

If both the predictors predict the same outcome, then there is no need to change the value of the corresponding saturating counter in the selector array. This situation basically means that we need to maintain status quo. However, if the outcomes are different, then the selector array's entry should be made to point towards that predictor, which gave the correct answer. If we assume that the states 00 and 01 correspond to $Pred_1$, then we need to decrement the saturating counter whenever $Pred_1$ is correct and $Pred_2$ is wrong. Likewise, we need to to do the reverse (i.e. increment the saturating counter), when $Pred_2$ is correct and $Pred_1$ is wrong. This mechanism ensures that we always choose the most accurate (and relevant) predictor for a given branch. This helps create the illusion that we have different branch predictors for different regions of code. Finally, the reason for choosing saturating counters is that we allow a certain amount of long term memory to be a part of the prediction process. We do not change predictors very frequently; we only do so when we make a given number of mistakes with the predictor that is currently chosen.

Tournament predictors have their fair share of overheads in terms of power, latency, and area. However, out of all the predictors that we have described, they are considered to be the most accurate, and are thus the predictors of choice in most cases.

Note that branch predictors in commercial chips use many more tables and combinations of bits, and use a hierarchy of different types of predictors. Discussing advanced designs is beyond the scope of this book.

3.4 Problem 3: What is the Target of a Branch?

Now, that we know if an instruction is a branch or not, and we can predict its outcome, let us predict the target of the branch. Let us here make a distinction between regular conditional and unconditional branches, and call/return statements. We claim that the latter two are special.

Consider the case of regular conditional and unconditional branches. Given the PC, we need to predict the target of the branch. In most ISAs, the address of the target is hard-coded into the instruction as a fixed offset with respect to the value of the PC. However, some ISAs support indirect branches where the target is contained in a register. In this case, predicting the branch target is more difficult.

Let us limit our discussion to cases where the offset is hard-coded in the instruction.

3.4.1 Branch Target Buffer (BTB)

Let us extend the IST to incorporate this information. We shall give it a new name and refer to it as the branch target buffer (BTB) henceforth. Let us add the branch target to each row of the BTB. The structure of the BTB is as shown in Figure 3.17. We use the least significant n bits of the PC to access a 2^n-entry BTB. Each entry of the BTB also contains the type of the branch (conditional/unconditional/call/return) and the target.

We can use the BTB for a dual purpose. It can be used to predict whether an instruction is a branch or not and the type of the branch. Furthermore, it can also be used to predict the target of a given branch.

To reduce the possibilities of destructive interference, we can adopt standard solutions as discussed in Section 3.2. Such solutions associate more information with each entry such that it is possible to differentiate between two branches that map to the same entry.

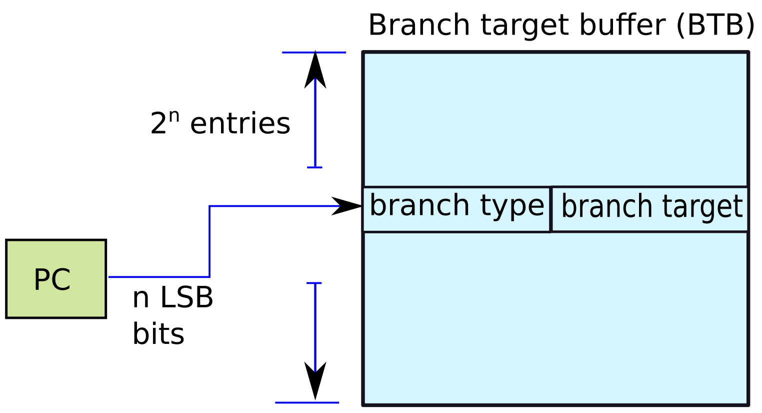

Figure 3.17: The branch target buffer

3.4.2 Call and Return Instructions

Let us now discuss the special case of function call and return instructions. They form a pattern. Consider the function call sequence shown in Figure 3.18.

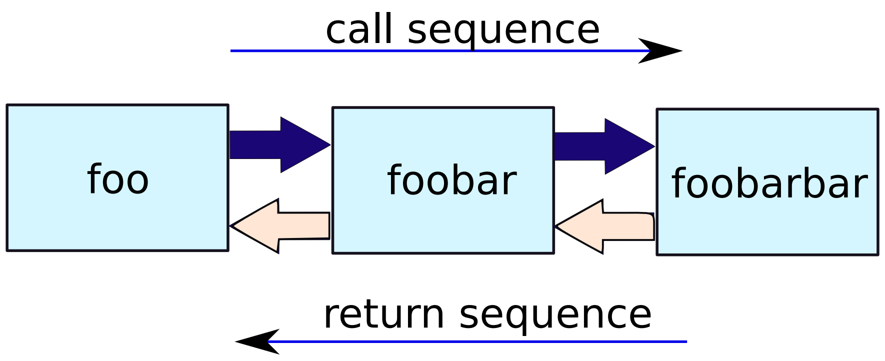

Figure 3.18: Call sequence ($foo \rightarrow foobar \rightarrow foobarbar$)

The function foo calls function $foobar$, which calls the function $foobarbar$. The return sequence is exactly the reverse $foobarbar \rightarrow foobar \rightarrow foo$. We can infer a last-in first-out behaviour. Computer architects and compiler writers have long exploited this pattern to optimise their designs. They start out with creating a stack of function calls. When we call a function we add an entry to the stack, and when we return from a function we pop the stack.

Let us thus create a stack of function call entries in hardware and refer to it as the *return address stack* (RAS). Each entry stores the return address (instruction immediately succeeding the function call) corresponding to the function call. Whenever, we encounter a call instruction, we insert its return address into the RAS. In our example, the state of the stack after calling the function $foobarbar$ is as shown in Figure 3.19.

Now, when we encounter a return instruction, all that we need to do is just pop the stack. Since we return from instructions in a strictly reverse order, this simple strategy will always work. For our example, when we fetch the return instruction for the function $foobarbar$, we can just pop the topmost entry from the RAS stack, and use it. The topmost entry contains the return address for this function.

Specifically, the algorithm is as follows. Given the PC, if we predict that the instruction is a return

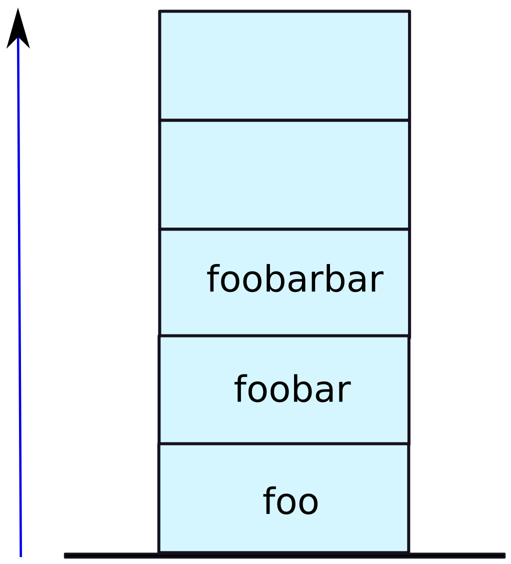

Figure 3.19: Function call stack

instruction, then all that we need to do is use the element at the top of the RAS stack as the branch target of the return instruction. After decoding the instruction, if we find that the instruction was genuinely a return instruction, we can then pop the stack and remove the entry.

Using the RAS stack for return instructions is a far more accurate method of predicting branch targets than the BTB, primarily, because we do not suffer from destructive interference. However, this strategy does have its limitations. The first is that like the BTB the RAS has a finite size. It is possible that we might find the size of the RAS to be insufficient especially if we have a chain of recursive function calls.

In spite of these issues, the RAS stack is regarded as one of the most accurate methods of predicting the return address in modern processors. To increase its accuracy, researchers have proposed modifications to take care of recursive function calls, and furthermore it is possible to move a part of the RAS to memory if there is an overflow.

3.5 The Decode Stage

The process of instruction decoding is essential to most processors. The decode unit accepts an instruction encoded in a binary format, expands it such that all the fields are readable, and are easily accessible to all the units that process the instruction. Specifically, the roles of the decode unit (also see [Sarangi, 2015]) are as follows:

1. Expand all the immediate values embedded in the instruction to full 32 or 64-bit numbers.

2. Extract the branch offset (offset from the current PC), and expand it to a 32/64-bit field. Sometimes the decode unit is also used to compute the branch target by adding the current PC to the offset.

3. Extract the ids of all the registers.

4. Sometimes instructions might have an implicit source. For example the *ret* (return from function) instruction might read a certain return address register. The decode unit can make this explicit at this stage by recording the id of the return address register as a source register.

5. The decode unit creates the *instruction packet*. The instruction packet is defined as a bundle of information that contains all the details regarding the processing of the instruction. This includes its opcode (type), ids of source registers, constants, offsets, and control signals. Control signals are used to control the behaviour of different functional units in the data path (units that perform data processing and data storage operations). They are also used to choose one among multiple inputs for functional units, particularly, when we are choosing between inputs read from the register file, and forwarded inputs from other stages. A basic textbook on computer architecture (such as [Sarangi, 2015]) discusses the difference between the control path and the data path in great detail. It also talks about the way that control signals are generated and used to control different elements in the data path, notably, multiplexers that choose between different inputs.

The decoder is typically one of the most complex logic blocks in the processor. It consumes a lot of power, and also takes up a lot of area. The design of the decoder is in general very specific to an ISA, and thus it is seldom a subject of advanced study. However, let us nevertheless discuss some general ideas on how to make the process of decoding more efficient.

When is decoding likely to be a big issue? Recall that one great advantage of RISC instructions is that they make the process of decoding very easy. As compared to RISC instructions, CISC instructions are typically very hard to decode. In fact, the decode complexity is often cited as one of the biggest reasons behind the reasons to choose a RISC ISA. Let us see how this situation can be made better for CISC processors in Section 3.5.1.

We can also make a few modifications to the decode stage to efficiently leverage some simple patterns in the code. These will be discussed in Sections 3.5.2 and 3.5.3.

3.5.1 Predecoding CISC Instructions

In this section we shall discuss a scheme originally proposed by Narayan et al. [Narayan and Tran, 1999].

Let us first understand the issues at hand. CISC instructions such as Intel's x86 can be very complex. As of 2020, we have 1000+ x86 instructions and the number is growing with every new generation of processors. Moreover, the length of the instructions is also not fixed. In fact, the length can be highly variable. An x86 instruction's length as of 2020 can vary from 1 byte to 15 bytes. Thus, even figuring out the boundaries of instructions is a non-trivial problem. We need to decode the instruction to find this information. This causes several problems.

In general, when we are fetching multiple instructions at once, we can improve the throughput by decoding them in parallel. This is very easy to do in a RISC ISA because each instruction is of the same size, and thus we consequently know the starting address of each instruction. However, doing this in a CISC ISA is very challenging. We in fact need to do this serially. We need to read an instruction and figure out its length in bytes before we can read the next instruction. This makes a slow decoder even more slower.

There are additional complications in modern CISC processors. Most structures within the processor are designed to work optimally when the ISA is homogeneous. Flexibility is the enemy of high performance and low power. Hence, most processors that use CISC ISAs convert CISC instructions into RISC instructions internally. They are often referred to as microinstructions or micro-operations(μops). The process of conversion of CISC instructions into μops is typically done by the decoder. It would also be useful to know about the number of μops a given CISC instruction translates to, such that we can reserve enough space in different CPU structures. Again, without decoding the instruction it is not possible to find this information.

To solve all of these problems, let us consider an alternative organisation. Assume that when a cache line enters the i-cache for the first time we make it pass through a predecoder. The predecoder scans the set of bytes and marks the instruction boundaries. In addition, it annotates the instruction with some more information such that the effort required to decode the instruction reduces. The patent by Narayan et al. [Narayan and Tran, 1999] suggests a method where we add 5 bits to every byte in an i-cache line. The overall scheme is shown in Figure 3.20.

The bits are as follows:

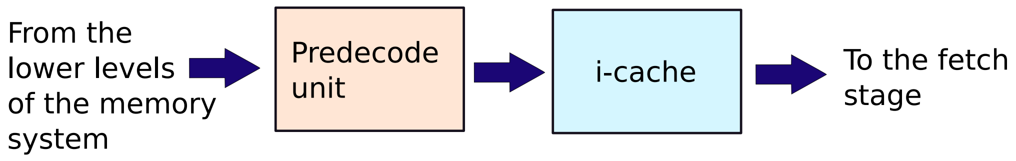

Figure 3.20: I-cache with a predecoder

1. A start bit: indicates if the byte is the starting byte of an instruction.
2. An end bit: indicates if the byte is the last byte of an instruction.
3. A functional bit that indicates the type of the instruction and the type of its operands. This is specific to the instruction.
4. Two bits named *two-ROP* and *three-ROP* respectively. Their meaning is interpreted in conjunction with the end bit. They indicate the number of μops in an instruction.

Note that this is only one of the possible ideas in this space. It is possible to annotate a cache line in the i-cache in other ways also. For example, we may just want to store the indices (offsets within the line) for the starting bytes of each instruction. The general principle is that we are at least storing two pieces of information: positions of the start and end bytes of each instruction and the number of μops required to implement a CISC instruction. This is enough for us to do many things. For example, if we know the lengths of the instructions, we can decode them in parallel. If we know the number of μops that each instruction translates to, we can reserve space for the microinstructions in different architectural structures.

To summarise, when we fetch an i-cache line, we read the instruction bytes as well as the extra information written by the predecoder. The fetch unit passes this information to the decode unit, which can almost instantaneously figure out the boundaries of instructions, and the number of μops that will be required to execute each instruction. Specifically, the hardware has many parallel units that scan the cache line starting from different points. They look for start and end bits. They extract all the instructions in their part of the cache line, and then start decoding them.

Thus, by expending a little bit of additional storage, we can parallelise the process of decoding CISC instruction sets.

3.5.2 Optimising Operations on the Stack Pointer

Let us consider one of the most commonly occurring patterns in a program. We typically have instructions of this form:

```
st r1, 12[sp]
...
ld r1, 12[sp]
```

In this case, we are storing a value into the stack (indexed by the stack pointer sp) from register $r1$ and later loading it back. Recall that this is a very common pattern and is most often used while saving and restoring the values of registers before and after a function call. Since a function can overwrite registers it is often necessary to store the values of registers (that stand to be overwritten) on the stack, and later restore them. We also use such kind of a pattern when we perform register spilling. Recall that register spilling refers to the situation where the compiler runs out of registers, and it becomes necessary to free up some registers by writing their values on to the stack. Later on, when the values are required, they can be read from the stack.

Load and Store Instructions

Given that this pattern is so common, we can perform some optimisations at the decode stage itself to accelerate the execution of such instructions. Figure 3.21 shows the scheme. It was originally proposed by Bekerman et al. [Bekerman et al., 2000]. We shall describe a simplified version of their scheme in this section.

We maintain a register in the decode stage called the current stack pointer, *csp*. It is initialised to the starting value of the stack pointer. Every time we read the stack pointer, a small adder adds the memory offset to the *csp* to compute the memory address of the memory instruction. Thus, the memory address for all such instructions is available at the decode stage itself. We shall see in subsequent chapters that there are many more stages in the pipeline that an instruction needs to go through before we can compute the memory address. This is thus a quick shortcut.

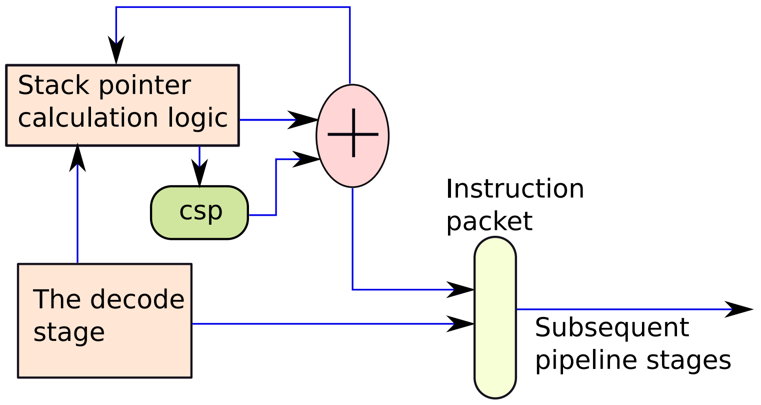

Figure 3.21: The stack pointer calculator

Refer to Figure 3.21 where we see that all we need is an adder to add the offset to the current value of the stack pointer. Since all the instructions till this point arrive in program order, we are sure that we are not reading wrong values.

Instructions that Update the Stack Pointer

Now, consider instructions that update the stack pointer. We typically have such instructions at the beginning and end of function calls. These instructions create an activation block (region in the stack) for the function to store its local variables, registers, and arguments. Before returning from a function, we destroy the activation block by updating the stack pointer. The stack pointer becomes the same as it was before the current function was called. These instructions are of the following type (assuming a downward growing stack).

```
/* create the activation block */
sub sp, sp, 24
...
...
/* delete the activation block */
add sp, sp, 24
```

We can add a quick shortcut to do this process as well. We can perform the same add or subtract operation on the contents of the *csp* register and update it. This process will ensure that the value of the stack pointer (stored in the *csp* register) is always up to date.

Tricky Corner Cases

Let us now consider an instruction of the type $\boxed{\textit{ld sp, 12[r1]}}$.

In this case, we are loading a value from memory and saving it in the stack pointer. We cannot possibly process this instruction in the decode stage. The value of *12[r1]* will only be available to us after the load instruction gets the value from the memory system. This pattern is rare, nevertheless, we need to account for it. In this case, the *csp* register will contain a *null* value indicating that it does not contain the value of the stack pointer. All subsequent instructions that use the stack pointer will be aware of the fact that the value of the stack pointer is not currently available. They will read the value of the stack pointer using the regular process that treats the stack pointer as any other register.

Nevertheless, we can also do something very smart here. Assume we have the following code snippet.

```
ld  sp, 12[r1]
add sp, sp, 24
...
add sp, sp, 12
```

In this case, the load instruction sets the value of the *csp* register to *null*. Hence, the subsequent add instructions do not get the correct value of the stack pointer in the decode stage. Instead of not doing anything, we record the changes made to the stack pointer. For example, after the first add instruction, we record the fact that the value of the current stack pointer exceeds the value read by the load instruction by 24. After the second add instruction, we note that the value of the stack pointer has increased by 36. We thus record the difference between the current stack pointer, and the stack pointer computed by the latest instruction that nullified the content of the *csp*. Let us refer to this as Δ.

When the load instruction returns, we set the contents of the *csp* to:

$$csp \leftarrow addr + \Delta$$

where *addr* is the value returned by the load instruction. We can then continue the process of using and updating the stack pointer in the decode stage.

Benefits of Computing the Stack Pointer's Value Early

Let us now look at the benefits of this process. We might not be in a position to understand all the benefits because we have not read about most of the OOO pipeline. Just to give a sneak peek of what lies ahead, we shall see that the OOO pipeline is very complicated and consists of many stages. The stage that computes the memory address might be 20-30 cycles away. It is thus extremely beneficial, if we get the address on the stack early. Let us list some of the advantages.

1. We can issue a load to the memory system based on the stack pointer's value early (in the decode stage itself). This will ensure that the corresponding data arrives early and can be used to write to the register. We also do not have to issue a separate load instruction to the memory system later on.

2. For both load and store instructions, we can completely omit the process of address calculation. We shall already have the address with us.

3. We can also get rid of the instructions that add and subtract fixed constants to the stack pointer. They need not be sent down the pipeline. This will free up space in different structures in later stages of the pipeline, and also reduce one instruction.

3.5.3 Instruction Compression

One of the major bottlenecks in the fetch process is that only 8 to 16 instructions fit in a 64-byte cache line. Sometimes, we need to fetch multiple cache lines if a set of instructions straddle cache line boundaries, or if we have "taken branches" in the code. If we can somehow fit more instructions in an i-cache line, we can sustain a higher fetch bandwidth. Let us look at two common approaches.

Reduced-Width Instructions

Let us start out by describing a scheme that is currently implemented in commercial processors. ARM instructions are typically 32-bit (4 bytes) instructions. In general, having large instructions increases their fetch and decode overheads. We need to understand that most programs do not use all the instructions with the same frequency and sometimes an instruction has multiple variants. We only use one or two variants. It is thus possible to define a much simpler and compact ISA that captures most of the frequently occurring patterns in programs.

Consider the 16-bit Thumb ISA that is a subset of the regular 32-bit ARM ISA [Sloss et al., 2004]. Thumb programs typically take up 35% less space as compared to regular programs written using the full 32-bit ARM ISA. Moreover, once they enter the pipeline, Thumb instructions are decompressed into full 32-bit instructions. There is no measurable loss in performance in this process, and in addition it is not necessary to change the internals of the processor to support Thumb instructions.

Without discussing the details of the ARM ISAs, let us look at the general principles underlying the creation of an ISA that has shorter instructions.

1. Reduce the number of instructions supported in the reduced-width ISA. We will require fewer bits to encode the opcode(type) of the instruction.

2. Avoid encoding complicated flags in each instruction such as condition codes (dependent on the outcome of the last comparison).

3. We can show a reduced view of the architectural registers. Instead of exposing all the architectural registers, we can expose a subset of the architectural register space to the reduced ISA. For example, the Thumb ISA only sees 8 general purpose registers as compared to the full scale ARM-7 ISA that has access to 16 general purpose registers. This helps us save 1 bit while encoding registers.

4. We can reduce the size of the immediate fields. This will reduce the size of the constants that we can embed in instructions. Such constants include branch and memory offsets. In most cases, we do not need very large offsets given that we have a high degree of temporal and spatial locality in most programs.

5. Use an implicit operand in an instruction. This means that one of the source registers is the same as the destination register. We shall thus have instructions of the form $\boxed{\textit{add r1, r2}}$.

   This translates to *add r1, r1, r2* ($r1 \leftarrow r1{+}r2$). In this case, the register $r1$ is known as an *accumulator*. To encode this instruction, we need to encode the ids of two registers instead of three (for most general purpose RISC ISAs). Using an accumulator reduces flexibility to some extent; however, since most ISA families have some support for accumulators (most notably the x86 ISA), there are very sophisticated compiler algorithms to generate code for ISAs with accumulators.

Compression Using Dictionaries

Let us consider an alternative approach. Let us identify frequently occurring sequences of instructions and replace them with a code word. At run time, we can retrieve the value of the code word in the decoder, and then retrieve the sequence of instructions by accessing a *dictionary*; it stores the mapping between the code word and the instruction sequence. This will increase the fetch bandwidth.

Let us discuss a scheme proposed by Lefurgy et al. [Lefurgy et al., 1997], albeit with some modifications. They propose a new pass after the compilation phase. Specifically, after the compiler has generated the binary, they introduce a pass where a program scans the instructions in the binary and figures out the most commonly used instruction sequences. It replaces each such sequence with a code word that uniquely identifies the code sequence. The mapping between the code words and the instruction sequences are saved in a dedicated region in the memory space of the program called the *dictionary*.

There are issues with regards to branches because now the code size changes. To keep things simple, the algorithm never encodes a branch instruction that uses relative offsets. Handling branch targets is complicated. If we are branching to a point that is in the middle of an encoded instruction sequence, then we need to specify the offset of the branch within the encoded instruction sequence. This creates an added layer of complexity. It is thus best to ensure that all branch targets are only at the beginning of encoded instruction sequences. This makes the scheme easy to use. Note that since the program effectively becomes shorter because instructions are now replaced with short code words, it is necessary to modify all the addresses that are used in the program, in specific, branch offsets. The post compilation pass needs to correct all of these offsets.

Once we run the program, and start fetching instructions, the first step is to see if we are fetching a regular instruction or a code word. If we are fetching a code word, then we need to access the dictionary, and fetch the instructions that correspond to the given code word. The subsequent stages of the pipeline run unmodified (see Figure 3.22).

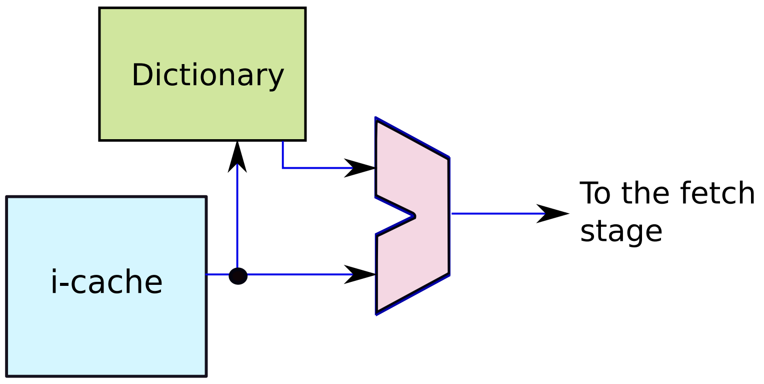

Figure 3.22: The fetch logic for a processor that uses compressed instructions stored in a dictionary

An astute reader may ask, "What is the benefit?" Previously we were reading uncompressed instructions directly from the i-cache, and now we are reading them from the dictionary. Let us mention the benefits.

1. The overall code size is lower. Assume one instruction sequence (containing 5 instructions) is found at 10 different locations in the program. Let the size of an instruction as well as that of a code word be 4 bytes. In this case, if we use the uncompressed version of the program we require $10 * 5 * 4 = 200$ bytes for these instructions. However, if we replace each instruction sequence with a 4-byte code word, then we require 20 $(4 * 5)$ bytes to store the sequence in the dictionary and $10 * 4$ bytes for the 10 corresponding code words in the program– one for each instance of the instruction sequence. We thus require a total storage space of 60 bytes. Given that the code size is lower, it will be easier to store such programs, particularly, in embedded systems where instruction memory is limited.

2. The dictionary can be stored on a separate storage structure, which is expected to be much smaller than the i-cache, and thus it can be much faster than the i-cache as well. If the encoded instruction sequence is long enough, then there will be a savings in terms of time if we fetch the constituent instructions from the dictionary.

3. We can extend this idea to store decoded instructions in the dictionary. We can thus effectively skip the decode stage using this optimisation.

3.6 Summary and Further Reading

3.6.1 Summary

Summary 2

1. *We need a high-throughput mechanism to deliver instructions to the pipeline. We typically need to deliver 2-4 instructions to the pipeline in a single cycle. This is known as the fetch width.*
2. *The first problem is to decide if a given instruction is a branch or not. We can keep a table that is addressed by a subset of the bits of the PC. This table can contain the type of the instruction including the type of the branch – conditional, unconditional, call, or return.*
3. *We refer to this table as the IST (instruction status table).*
4. *The second problem is to predict the direction of a branch – taken or not-taken. The simplest method is to use the last n bits of the PC to access a table of bits. Each bit remembers the outcome of the branch when it was encountered for the last time. This is called a Bimodal Predictor.*
5. *Any such predictor that uses bits of the PC for addressing can suffer from the problem of aliasing or destructive interference. This refers to a phenomenon where multiple entries map to the same entry. A standard approach to solve such issues is to store additional bits of the PC (referred to as the tag) or to create a set associative structure. The latter is a standard technique used in the design of caches (small instruction or data memories) where we associate a given PC with a set of locations in a table. It can be present in any of the locations within the set. This gives us more flexibility.*
6. *We can further augment the bimodal approach by having a table of saturating counters. Each such counter incorporates a certain degree of hysteresis. One wrong prediction does not change the outcome of the predictor.*
7. *We can further improve the accuracy of these predictors by incorporating global history where we consider the behaviour of the last few branches. This often determines the context of a prediction.*
8. *It is possible to further generalise this idea by considering the full design space of predictors that include different degrees of local history (same branch across time) and global history (last k branches in the same time window). They are referred to as the GAg, GAp, PAg, and PAp predictors.*
9. *The GShare predictor tries to reduce the amount of storage and still perform as well as a PAp predictor by indexing the table of saturating counters with a XOR of bits from the PC and the branch history.*
10. *The tournament predictor is composed of multiple predictors. The final outcome is equal to the outcome of the most accurate constituent predictor.*
11. *We can enhance the IST to create a structure called the branch target buffer (BTB) that also stores the branch target.*

12. *For return instructions, we prefer not to use the BTB mechanism. Instead it is a better idea to use a stack of return addresses known as the RAS (return address stack).*
13. *The fetch unit feeds its instructions to the decode unit. The decode unit is in general very simple for RISC ISAs. However, in variable length CISC ISAs, the decoder is very complex. It is often necessary to annotate cache lines with information to demarcate instruction boundaries.*
14. *We can do several decode time optimisations such as computing the value of the stack pointer in the decode stage.*
15. *We can also compress instructions offline, and dynamically decompress them at the time of execution. This will save us valuable space in the i-cache.*

3.6.2 Further Reading

The original papers on two-level branch predictors that incorporated global history were published by Yeh and Patt [Yeh and Patt, 1991, Yeh and Patt, 1992, Yeh and Patt, 1993]. Some advanced branch prediction mechanisms include the agree predictor [Sprangle et al., 1997], the YAGS predictor [Eden and Mudge, 1998], and the TAGE predictor [Seznec, 2007]. The paper on Alpha EV8 [Seznec et al., 2002] gives a perspective on branch predictors implemented in commercial processors.

Recent approaches have focused on novel methods for predicting branches. One of the most promising directions is based on neural networks [Jiménez, 2003, Seznec, 2004, Jiménez, 2011b]. Some of them even use analog electronics [Jiménez, 2011a] and memristors [Wang et al., 2013]. A recent survey by Mittal contains many more references of papers on branch predictors [Mittal, 2018] and also a detailed analysis of the design space.

There are some works that map the problem of branch prediction to predicting general sequences. Using information theoretic measures they derive an error bound, and also correlate this with the compressibility of the sequence (see [Federovsky et al., 1998]).

The area of instruction compression is very well studied. Some of the popular papers in this area are [Benini et al., 1999, Chen et al., 1997, Helkala et al., 2014].

Exercises

Ex. 1 — Show the working of a predictor with a 3-bit saturating counter.

Ex. 2 — How does a GShare predictor combine the PC bits and the branch history? What are the advantages of doing so? Do you expect it to be as effective as a PAp predictor?

Ex. 3 — Let us consider a tournament branch predictor with the following design.

- It contains an array of 3-bit saturating counters to choose between two predictors: $predictorA$ and $predictorB$. It uses the last n bits of the PC address to index this table.
- $predictorA$ is a simple branch predictor that uses m bits of the PC to access an array of 2-bit saturating counters.
- $predictorB$ is a global branch predictor that us the last k branches to access a table of 2-bit saturating counters.

How many bits are used in total?

Ex. 4 — Design a method to predict the targets of indirect branches. An indirect branch stores the branch target in a register.

Ex. 5 — Consider the code for regular matrix multiplication. What is the best branch predictor for this code pattern?

Ex. 6 — Is it a good idea to have saturating counters with 3 or 4 bits?

Ex. 7 — Design a two-level branch predictor with the following property. The branch predictor at the first level produces a confidence along with a prediction. If the confidence is low, then the more elaborate branch predictor at the second level is used. This design can save power if used correctly.

Ex. 8 — We have used the LSB bits of the address to access the branch predictors. Even if we have multiple predictors, this can cause destructive interference in all the predictors. Instead of using the LSB bits, can we use different hashing functions to map the PC with entries in the branch predictors? Comment on this design choice.

* **Ex. 9** — Let us say that we take a regular branch predictor and augment it with a biased coin that yields Heads with probability p. Every time that we need to make a prediction, we flip the coin, and if we get Heads, then we flip the prediction of the branch predictor. In the general case, will this design lead to a better prediction?

* **Ex. 10** — Show the detailed design of a fetch unit that can predict more than 1 branch per cycle. Explain the trade-offs.

* **Ex. 11** — Consider a structure called a *loop buffer*. This contains the decoded instructions inside a loop. While executing the instructions in a loop, the processor gets the instructions from the loop buffer instead of the i-cache. Answer the following questions.

1. How do we detect a loop in hardware?
2. How do you think the loop buffer works? Provide details of its design and operation.
3. What should the size of the loop buffer and associated structures be?
4. What are the advantages of a loop buffer?

** **Ex. 12** — Consider an OOO pipeline where the branch predictor takes 2 cycles (instead of 1). The BTB however takes just 1 cycle. How can we ensure back-to-back execution of instructions (including branches) in such a pipeline. Note that back-to-back execution means that consecutive instructions can be fetched and executed in consecutive cycles. Show all the details of your proposed solution, and prove its correctness. [HINT: We should be prepared to do some extra work that might potentially get wasted.]

Ex. 13 — What are the advantages of predecoding CISC instructions?

* **Ex. 14** — Is recursion a desirable feature in programs in the context of the Return Address Stack (RAS)? How can we make the RAS aware of a recursive pattern in the program? Can you propose an optimisation for the RAS when our workloads have a lot of recursive function calls.

Design Problems

Ex. 15 — Understand the implementation of branch predictors in an architectural simulator.

Ex. 16 — Implement a branch predictor based on neural networks in an architectural simulator.

Ex. 17 — Implement an instruction compression scheme in an architectural simulator.

Ex. 18 — There is an intimate connection between compressibility of a sequence and the predictability. This is captured by the Fano's inequality. Use it to find the upper bound on the prediction accuracy of branches for different workloads.