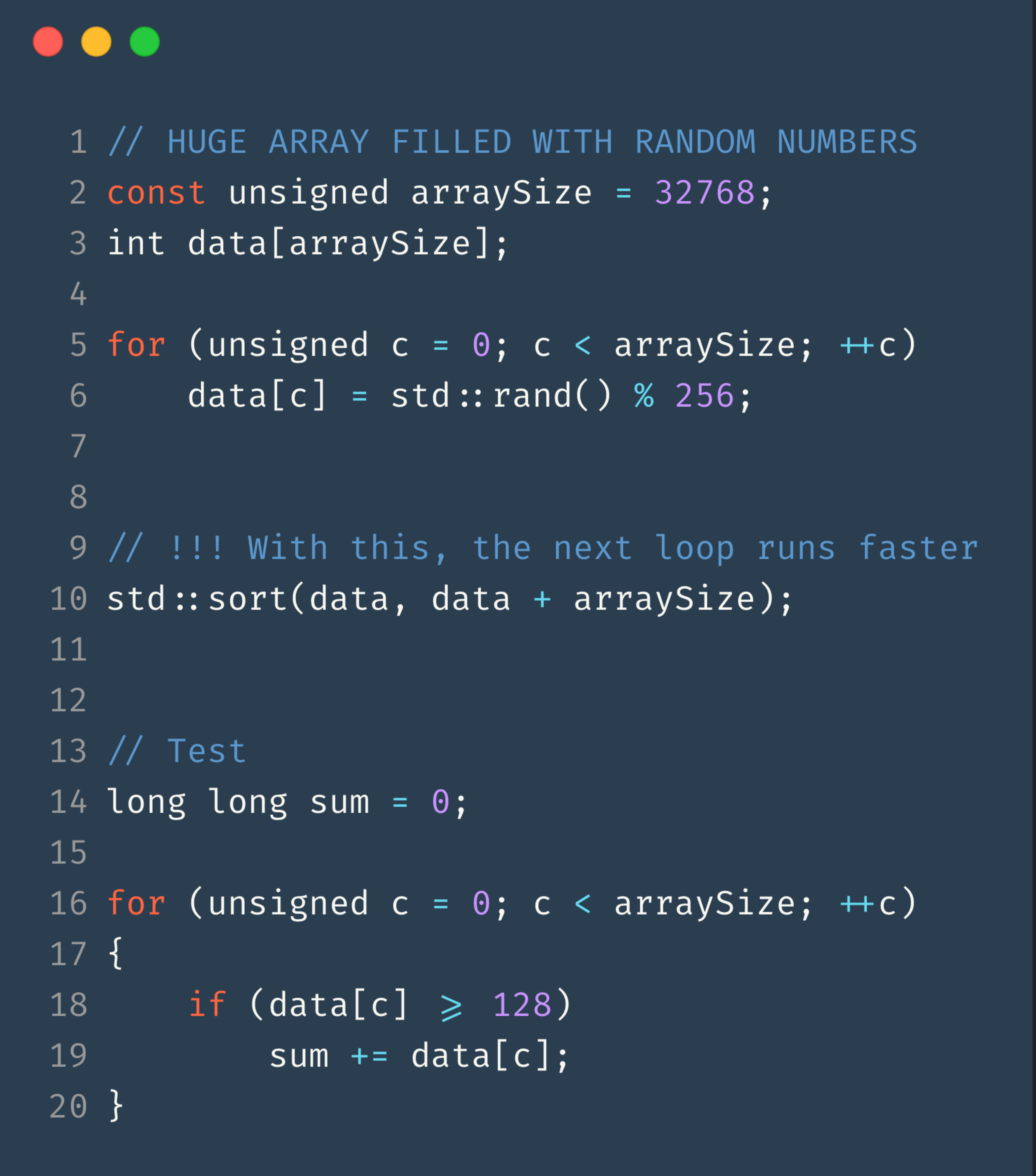
HORDON13

BRANCH PREDICTION

WHAT IS THE MOST UPVOTED QUESTION ON STACK OVERFLOW?

WHY IS IT FASTER TO PROCESS A SORTED ARRAY THAN AN UNSORTED ARRAY?

23 047 UPVOTE



```cpp
// HUGE ARRAY FILLED WITH RANDOM NUMBERS
const unsigned arraySize = 32768;
int data[arraySize];

for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;


// !!! With this, the next loop runs faster
std::sort(data, data + arraySize);


// Test
long long sum = 0;

for (unsigned c = 0; c < arraySize; ++c)
{
    if (data[c] >= 128)
        sum += data[c];
}
```

2.3 GHZ INTEL CORE I5

- **WITHOUT SORT THE RUNTIME IS:**

  19.7257 SEC

- **WITH SORT THE RUNTIME IS:**

  6.39513 SEC

BRANCH PREDICTION

- you = operator of a junction
- train is coming, but you have no idea which way it is supposed to go
- stop the train and ask the driver
- **Trains are heavy and have a lot of inertia. So they take forever to start up and slow down.**

IS THERE A BETTER WAY?

BRANCH PREDICTION

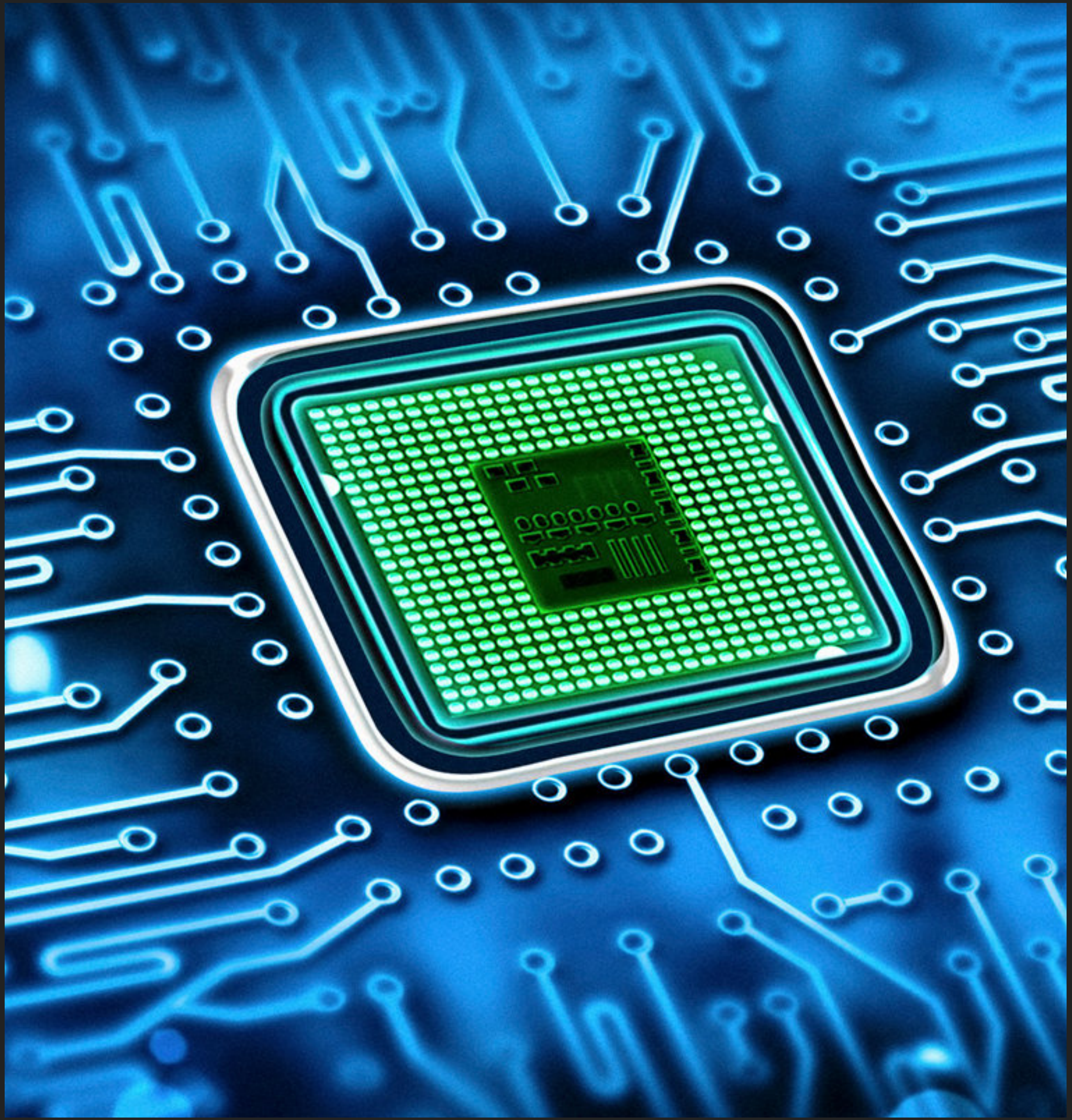
- Guess which direction the train will go!
- Two possibilities:
  - If you guessed right, it continues on. = **faster** process
  - If you guessed wrong, the captain will stop, back up, and yell at you to flip the switch. Then it can restart down the other path = **slower process**

DOUBLE OR NOTHING…

BRANCH PREDICTION

- You are a processor and you see a branch.
- Option 1: You halt the execution and wait until the previous instructions are completed the go the rigth path.
- Option 2: You guess which direction the branch will go…

-» BRANCH PREDICTION



BASED ON THE PREVIOUS DECISIONS

THE PROCESSOR TRY TO IDENTIFY A PATTERN AND FOLLOW IT…

T = branch taken

N = branch not taken

SORTED = EASY TO PREDICT

data[] = 0, 1, 2, 3, 4, ... 126, 127, 128, 129, 130, ... 250, 251, 252, ...

branch = NNNNNNNNNNNNN ... NNNNNNNTTTTTTTTTT ... TTTTTTTTTT

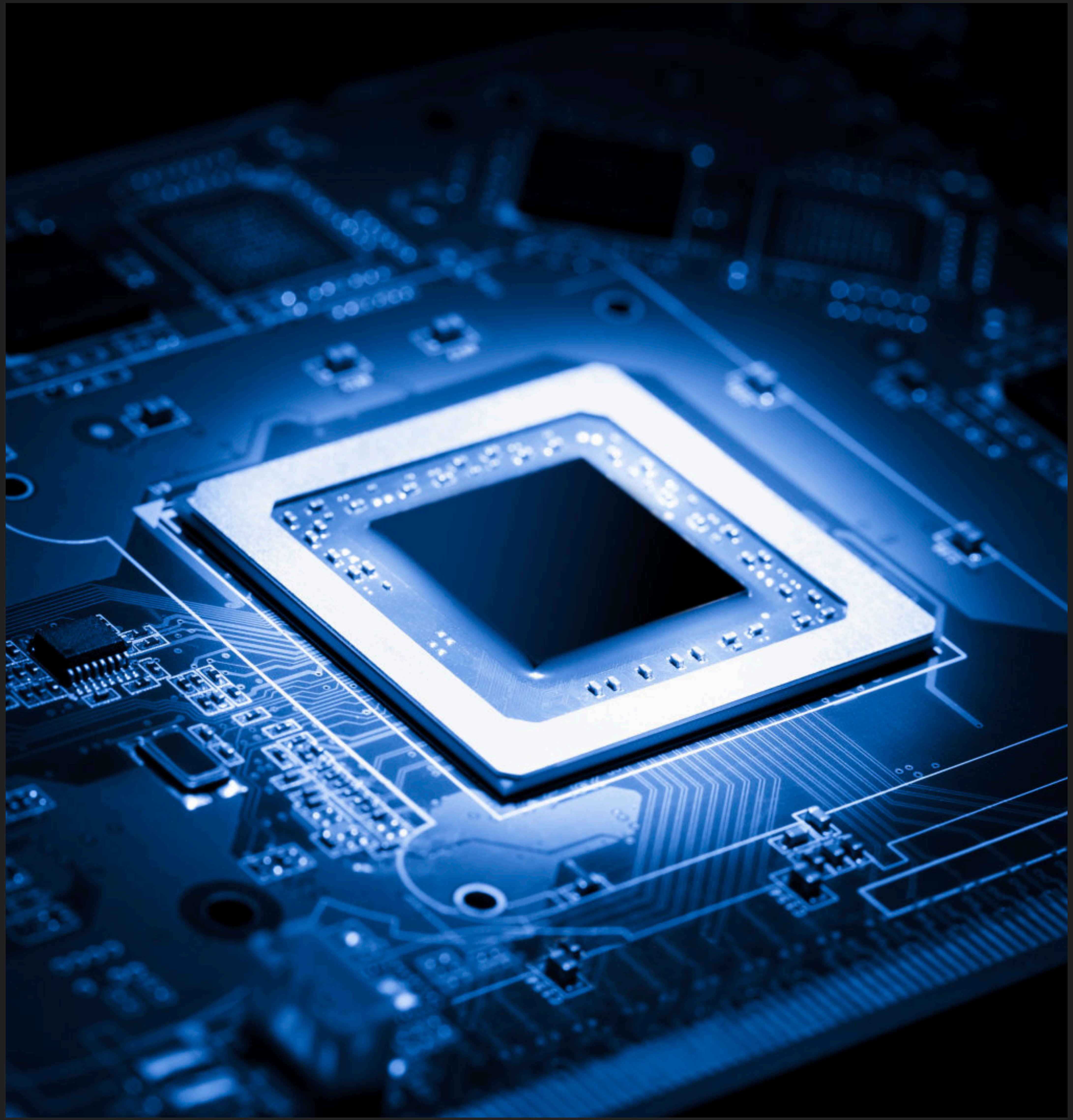
NOT SORTED = HARD TO PREDICT

data[] = 226, 185, 125, 158, 198, 144, 217, 79, 202, 118, 133, ...

branch = TTNTTTTNTNNTTTNTTNTTTTNTNNTTTNTTNTTTTNTNNTTTN

SOME INTERESTING TIDBIT…

- Most code has easy to predict branches so the modern branch predictors has **~90%** accuracy
- The same codes' runtime in Java (for example) has less differencies
  - because compliers can optimalize out some branches into **conditional move** (assembly: cmovl)



if condition = branch

```
if (data[c] >= 128)
    sum += data[c];
```

```
    add         eax, ebx
    cmp         eax, 0x10
    je          .skip
    mov         ebx, ecx
.skip:
    add         eax, ecx
```

no pipeline

ternary operator = no branch

```
sum += data[c] >=128 ? data[c] : 0;
```

```
add         eax, ebx
cmp         eax, 0x10
cmovne      ebx, ecx
add         eax, ecx
```

pipeline

So why does a conditional move perform better?

In a typical x86 processor, the execution of an instruction is divided into several stages. Roughly, we have different hardware to deal with different stages. So we do not have to wait for one instruction to finish to start a new one. This is called pipelining.

In a branch case, the following instruction is determined by the preceding one, so we cannot do pipelining. We have to either wait or predict.

In a conditional move case, the execution conditional move instruction is divided into several stages, but the earlier stages like Fetch and Decode does not depend on the result of the previous instruction; only latter stages need the result. Thus, we wait a fraction of one instruction's execution time. This is why the conditional move version is slower than the branch when prediction is easy.