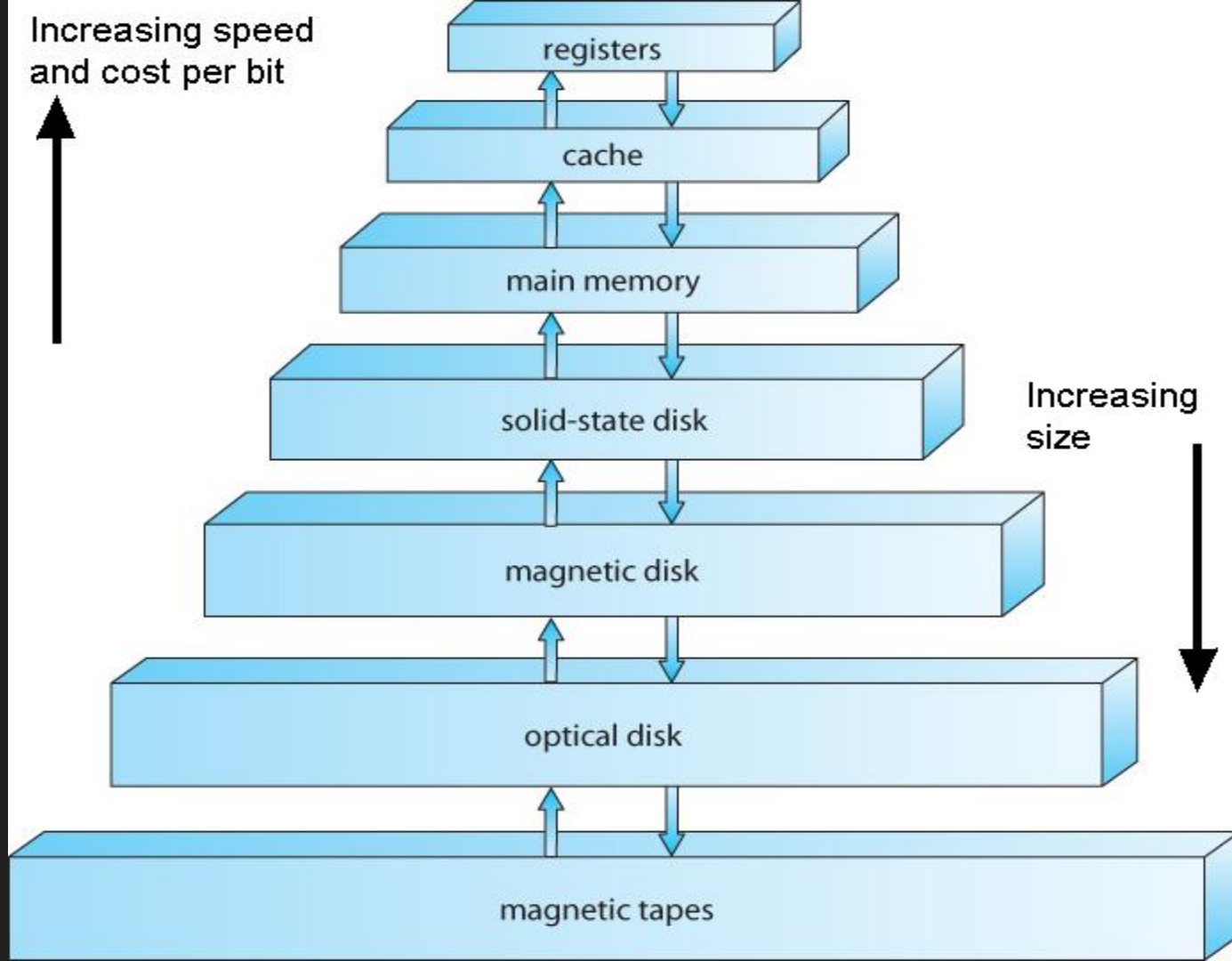


Performance programming

Memory lookup times

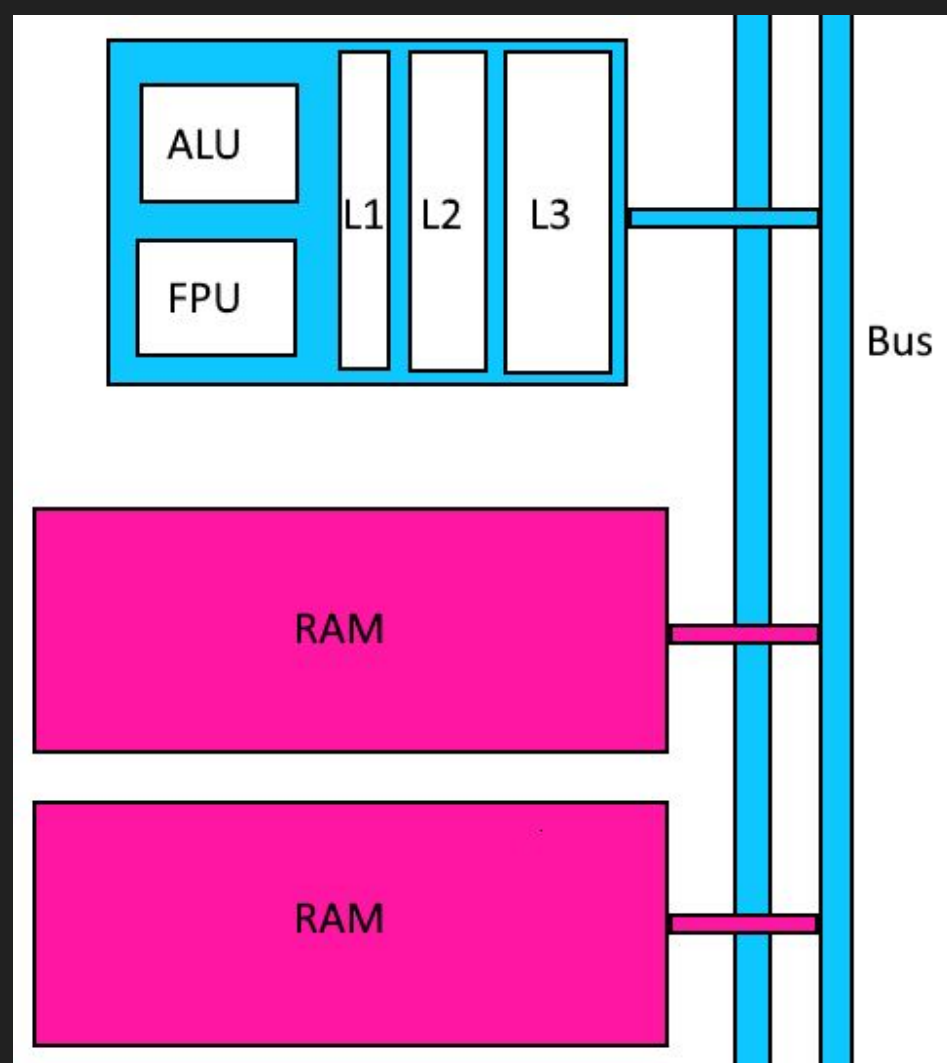
Action	Time taken	If one cycle was a second
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50000-150000 ns	2-6 days
Rotational disk I/O	1,000,000-10,000,000 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years
OS virtualization reboot	4 s	423 years
SCSI command time-out	30 s	3000 years
Hardware virtualization reboot	40 s	4000 years
Physical system reboot	5 m	32 millenia

Increasing speed
and cost per bit



Why should you care?

- Ever Heard of a cache miss?
 - Temporal locality
 - spacial locality
 - Applies to data & instructions
- Branching
 - Branch prediction/miss-prediction



Temporal locality

- Data is kept in the cache because it has been used recently and the processor “thinks” that it will be needed again soon.
- Reusing the same data will cost 14-200x less than using new data.

```
private void button1_Click(object sender, EventArgs e)
{
    int answer = 0;

    for(int i = 1; i < 101; ++i)
    {
        answer = answer + 1;
    }

    MessageBox.Show(answer.ToString());
}
```

Spatial locality

- The processor loads items it “thinks” will be used soon, based on what has been used already.
- extremely effective for arrays (which are very common)

Name	roll[0]	roll[1]	roll[2]	roll[3]	roll[4]	roll[5]	roll[6]	roll[7]
Values	12	45	32	23	17	49	5	11
Address	1000	1002	1004	1006	1008	1010	1012	1014

1-D Array memory arrangement

How can we use this knowledge to our advantage?

- Have a small working set!
- Do not make copies of variables if you can avoid it.
- Prefer linear access instead of jumping around in the array.
- Loops with obvious access patterns, such as every 2,3,4...n elements.

Example

```
int clearScreen(int[] pixelArray, int screenWidth,  
               int screenHeight, int clearColour)  
{  
    for(int x = 0; x < width; ++x)  
        for(int y = 0; y < height; ++y)  
            pixelArray[x + screenWidth*y] = clearColour;  
}
```

VS

```
int clearScreen(int[] pixelArray, int screenWidth,  
               int screenHeight, int clearColour)  
{  
    for(int y = 0; y < height; ++y)  
        for(int x = 0; x < width; ++x)  
            pixelArray[x + screenWidth*y] = clearColour;  
}
```


Branching and the Pipeline!

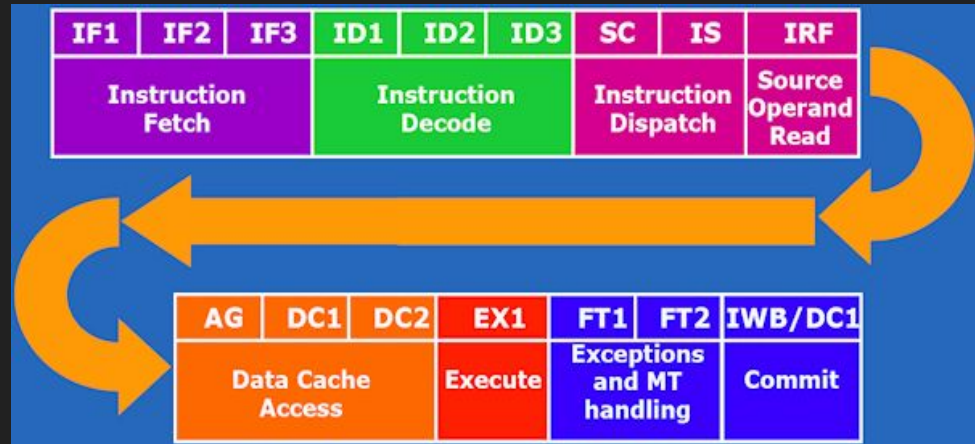
- A branch occurs on a conditional, such as an if/switch statement.
- A branch is just the name given to the specific decision the processor makes about which outcome of your conditional to execute.

```
int rand = Math.random()*100;  
  
if(rand<50)  
{  
    heads();  
}  
else  
{  
    tails();  
}
```



Pipelining

- Each stage requires about 1 clock cycle. Processors are measured in Ghz. Hz means clock cycles per second. so a 3Ghz cpu could execute about 3 billion clock cycles per second.
- Instructions vary per processor, but each line of Assembly language requires a full pipeline.
- pipelines can be up to 30+ clock cycles long.



How does this help?

Bit hacking for common operations:

The minimum of 2 integers:

```
if (x < y)
r = x;
else r = y;
```

There's a 50/50 that this will do a branch mispredict, since the hardware cannot know for sure what x or y is. The cost of a branch mispredict can be the length of the pipeline. While the following does not work in Java, in C++ it yields about a 15% performance increase.

```
r = y ^ ((x ^ y) & -(x < y)); //min of x,y
r = x ^ ((x ^ y) & -(x < y)); //max of x,y
```

On the same machine this will hit 4 clock cycles every time.

So if you are re-using this code n times, then it will average out at about 2x of the pipeline cycles+whatever it costs for the operations for the conditionals. vs 1 pipeline everytime. which is a rough 50% improvement in speed in the worst case.

Modular addition:

$r = (x + y) \% n$; This is expensive because division (except by 2) is expensive.

$z = x + y$;

$r = (z < n) ? z : z - n$; This is expensive because of an unpredictable branch.

$z = x + y$;

$r = z - (n \& -(z \geq n))$; The same trick as the minimum.

Compute $2^{\lceil \log n \rceil}$: (checking if the number n is to the power of 2)

--n;

$n |= n >> 1$;

$n |= n >> 2$;

$n |= n >> 4$;

$n |= n >> 8$;

$n |= n >> 16$;

$n |= n >> 32$;

++n;

More Bit Hacks:

Find the value of the least significant bit of any integer:

```
r = x & (-x);
```

Count the number of 1 bits in a word:

```
// Create masks
B5 = !((-1) << 32);
B4 = B5 ^ (B5 << 16); Performance is O(log(n))
B3 = B4 ^ (B4 << 8);  where n = word length
B2 = B3 ^ (B3 << 4);
B1 = B2 ^ (B2 << 2);
B0 = B1 ^ (B1 << 1);
// Compute popcount
x = ((x >> 1) & B0) + (x & B0);
x = ((x >> 2) & B1) + (x & B1);
x = ((x >> 4) + x) & B2;
x = ((x >> 8) + x) & B3;
x = ((x >> 16) + x) & B4;
x = ((x >> 32) + x) & B5;
```

Missing No. A case study




What was it?

1. Go to viridian city and talk to the OLD MAN.
2. Wait for him to finish his tutorial.
3. Fly to Cinnabar island.
4. Surf the coast by the gym, making sure you never leave the coast.

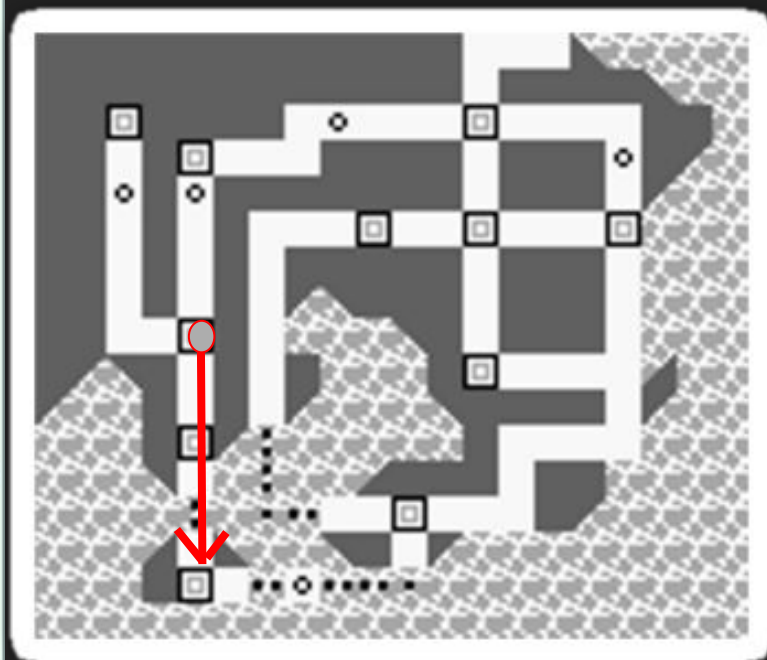
WEEDLE

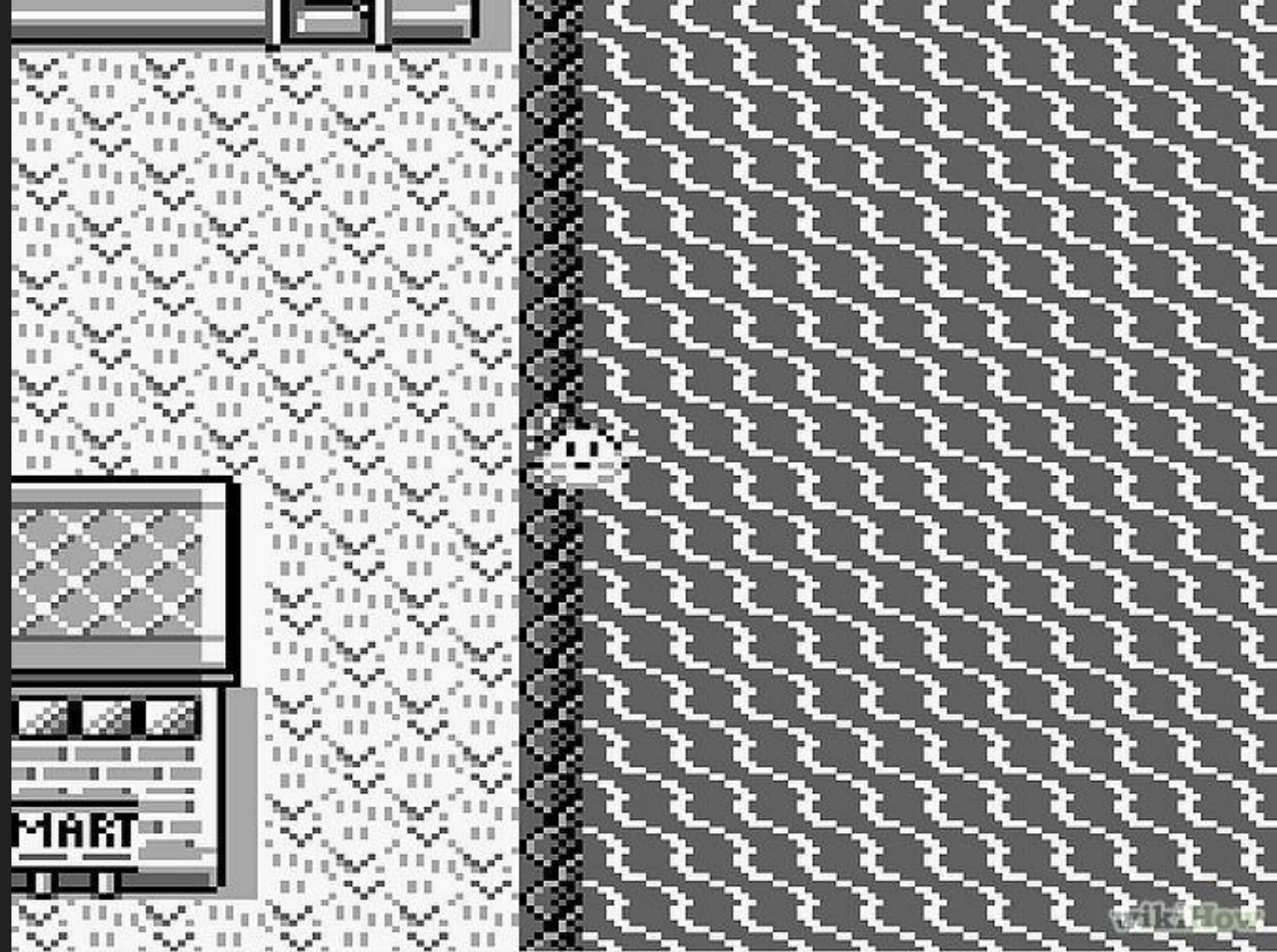
:L5

HP: 



OLD MAN used
POKé BALL!







Luis Neto
@ spidersaiyan
-2015-

Why does all this happen?

- When the OLD MAN comes onto the screen, it displays the OLD MAN's name. In order to do this, the game needs to temporarily change your name to OLD MAN. Instead of placing your name in a temporary variable, which would have consumed precious memory, the game stores your name in the Pokémon walk encounter table. That's where the fun begins.
- Normally, this would cause no abnormal activity, as this data is overwritten when the player moves to a different area.
- In all cities, however, this data remains blank, and so the data is never overwritten (as there is nothing new to overwrite it with), and thus, the data that was last entered (be it the player's name or the wild Pokémon data from another area) remains in place. This itself still causes no harm; however, an oversight in the programming of the tiles used to denote the shore of Cinnabar Island marks them as equivalent to grass. As all water routes have no real grass on them, likewise, the data is not overwritten, and so whatever data is in the slots for wild Pokémon found in the grass is used, be now it's the player's name (or a wild Pokémon found elsewhere, such as the Safari Zone).
- Other implications included the buffer overflowing into the bag slot 6, allowing you to get the maximum number of items for it. This showed as infinite, but really the game would only show up to 128. The overflow forced the buffer to be it's maximum(65k, since they used a short to store the variable, or rather a 16bit, 2byte variable was the standard. now a 64bit 8 byte variable is becoming standard.)

A visualisation:



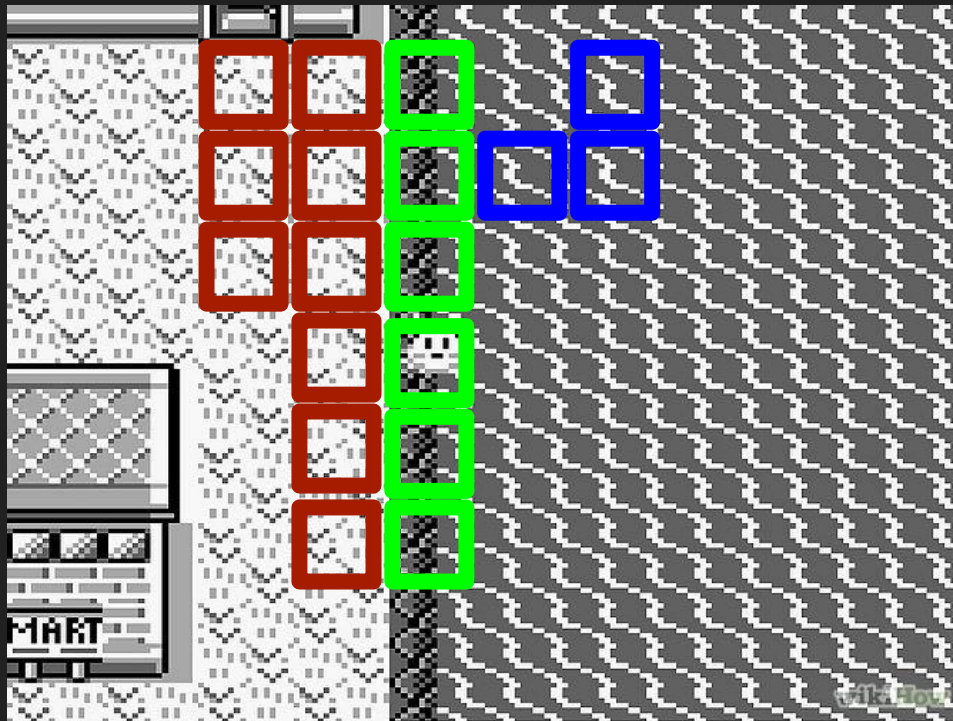
= City based tile, data is not overwritten since the developers did not expect to find pokemon in cities.



= Grass based tile. buffer uses whatever has been assigned to it last. Though that usually happens at fixed points in the game, such as area changes.



= Water based tile, the transition between grass and water based tile overwrites the pokemon encounter buffer. Meaning you will encounter normal pokemon.



- The Issue was that the Encounter buffers were overwritten at this point:

- When they should have been overwritten at the intersection of brown to green.

