MAME CPS2 Graphics Format

CPS2 B Board and MAME

CPS2 graphics in the MAME ROM format *mostly* mirror how the data is stored on the EEPROMs on an actual CPS2 B board.

In the case of `vsav`, there are eight 32 megabit (or 4 MB) files, each of which correspond to the respective EEPROM slot on the B board, for a total of 32 megabytes of graphics data.

Parsing the MAME Graphics Definition

From MAME's ROM definition for `vsav`:

```
ROM_REGION( 0x2000000, "gfx", 0 )
ROM_LOAD64_WORD( "vm3.13m", 0x0000000, 0x400000 )
ROM_LOAD64_WORD( "vm3.15m", 0x0000002, 0x400000 )
ROM_LOAD64_WORD( "vm3.17m", 0x0000004, 0x400000 )
ROM_LOAD64_WORD( "vm3.19m", 0x0000006, 0x400000 )
ROM_LOAD64_WORD( "vm3.14m", 0x1000000, 0x400000 )
ROM_LOAD64_WORD( "vm3.16m", 0x1000002, 0x400000 )
ROM_LOAD64_WORD( "vm3.18m", 0x1000004, 0x400000 )
ROM_LOAD64_WORD( "vm3.20m", 0x1000006, 0x400000 )
```

Some attributes have been omitted for lack of relevance.

MAME is loading each file in this order, but what does any of this mean?

MAME defines `ROM_LOAD64_WORD`:

```
#define ROM_LOAD64_WORD(name,offset,length) ROMX_LOAD(name, offset, length,
ROM_GROUPWORD | ROM_SKIP(6))
```

So the arguments are `name`, `offset`, and `length`, respectively, but what are those flags at the end?

Without digging much deeper into the code, `ROM_GROUPWORD` groups bytes in the region in pairs, otherwise known as `WORD`s.

`ROM_SKIP` defines how many bytes to skip after each group.

The implication is that it's grouping bytes in words and then skipping 6 bytes. That's likely why it's called `ROM_` ***`LOAD64_WORD`***; it's loading a 2 byte chunk, hence the `WORD`, and then skipping 6 bytes,

meaning it's working in 8 byte chunks at a time. Eight bytes is 64 bits, hence the `LOAD64`.

> **⚠ Attention**
>
> Different ROM formats load this data differently. The information here specifically applies to CPS2 ROMs. Even CPS1 is different.

Translated to TypeScript in a perhaps more human readable format:

```
vsav: {
  gfx: {
    size: 33554432,
    operations: [ {
      offset: 0,        length: 4194304, group: 2, skip: 6, file: 'vm3.13m'
    }, {
      offset: 2,        length: 4194304, group: 2, skip: 6, file: 'vm3.15m'
    }, {
      offset: 4,        length: 4194304, group: 2, skip: 6, file: 'vm3.17m'
    }, {
      offset: 6,        length: 4194304, group: 2, skip: 6, file: 'vm3.19m'
    }, {
      offset: 16777216, length: 4194304, group: 2, skip: 6, file: 'vm3.14m'
    }, {
      offset: 16777218, length: 4194304, group: 2, skip: 6, file: 'vm3.16m'
    }, {
      offset: 16777220, length: 4194304, group: 2, skip: 6, file: 'vm3.18m'
    }, {
      offset: 16777222, length: 4194304, group: 2, skip: 6, file: 'vm3.20m'
    } ]
  },
  // ...
}
```

> **ⓘ Info**
>
> The `offset` values here are only 2 bytes apart. This is because the data is interleaved, which is why it is writing 2 bytes and then skipping 6; it is interleaving all of the files.

MAME Graphics Processing

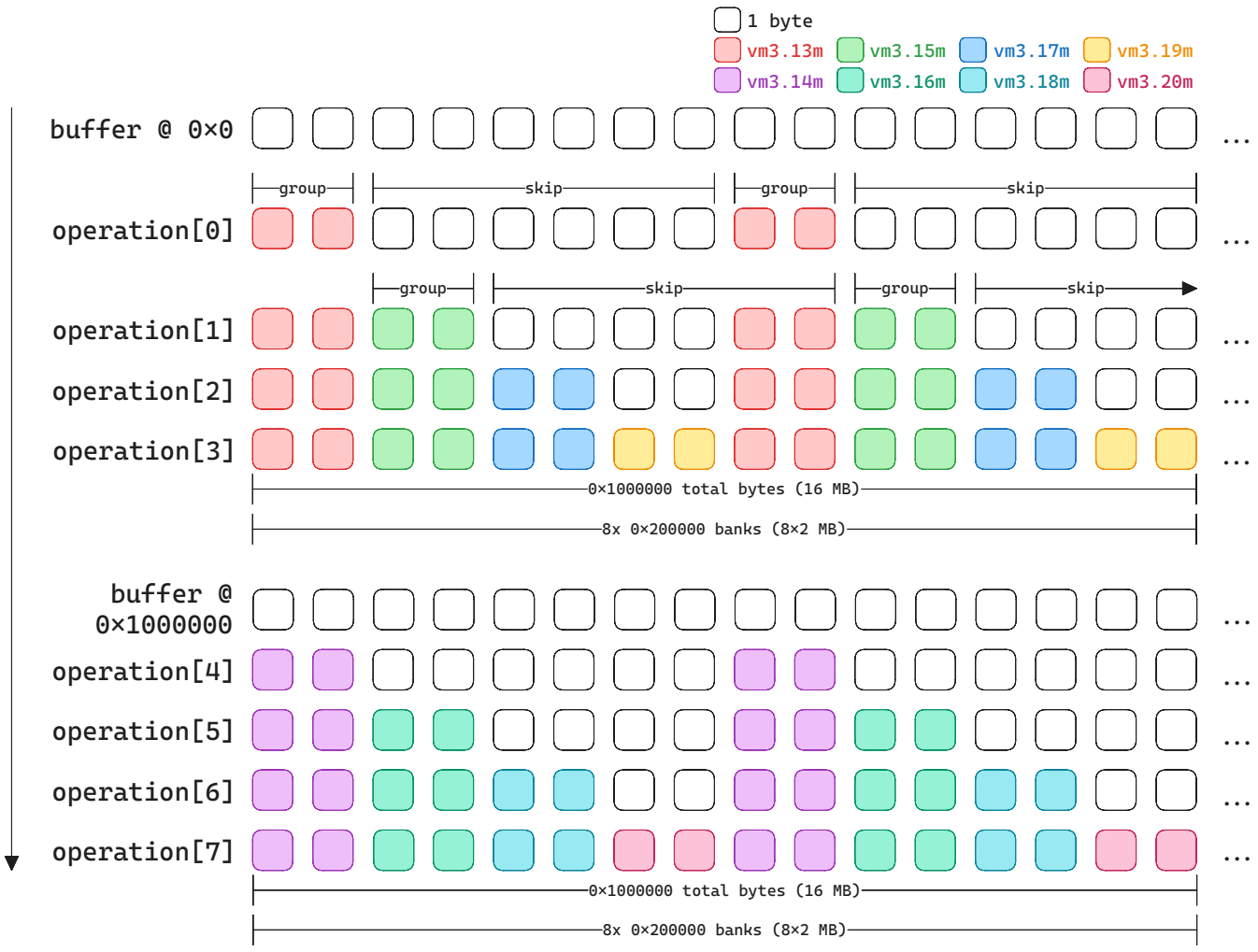
Interleaving Graphics Data

First, before any of the operations are performed, a buffer of the size of the graphics region is allocated.

Then, each operation is performed, loading each `file` into the buffer starting at `offset`, writing `group` bytes at a time, then skipping `skip` bytes. In this case, this means the first operation starts at offset `0`, writes `2` bytes from `vm3.13m`, skips `6` bytes, writes `2` more bytes from `vm3.13m`, and so on, until `length` bytes have been written.

It then moves on to the next operation (and, consequently, file) and repeats this until there are no more operations left.

Below is a graphical representation of what happens to the first 16 bytes of the buffer as each of the operations are performed.



> **Info**
>
> The second 4 operations are at a much larger offset. This is because the first 4 operations occupy the first half of the 32 MB space while the second 4 operations occupy the second half of the 32 MB space.

Decoding Graphics Data

MAME next decodes the graphics data using the code and algorithm below.

```
void cps2_state::cps2_gfx_decode()
{
        const int banksize = 0x200000; // 2 MB
        const auto size = memregion("gfx")->bytes();

        for (int i = 0; i < size; i += banksize)
                unshuffle((uint64_t *)(memregion("gfx")->base() + i), banksize /
8);
}

void cps2_state::unshuffle(uint64_t *buf, int len)
{
        if (len == 2)
                return;

        assert(len % 4 == 0);   /* must not happen */

        len /= 2;

        unshuffle(buf, len);
        unshuffle(buf + len, len);

        for (int i = 0; i < len / 2; i++)
        {
                const uint64_t t = buf[len / 2 + i];
                buf[len / 2 + i] = buf[len + i];
                buf[len + i] = t;
        }
}
```

> **Note**
>
> For `vsav`, `size` will be `0x2000000` (32 MB).

The data is cut up into 2 MB banks, then these banks are iterated through and "unshuffled" using the `unshuffle()` function.

This algorithm can be somewhat difficult to scrutinize, as it's implemented recursively and modifies the data *in-place* (that is, the original data is modified while the algorithm runs on it), but essentially, for each of the banks, the `unshuffle()` operation is called.

`unshuffle()` breakdown

Each bank is operated on in a 64 bit buffer that also happens to be the original buffer of the graphics binary data. This means the graphics data itself is altered as `unshuffle()` runs.

> **Info**
>
> `banksize / 8` is passed as `len` because `banksize` is an 8 bit measurement while the units being operated on are 64 bit

The buffer is broken in half, and then each half is broken in half, and so on, until every piece is length 2. Then, each latter half of each piece is swapped with each former half of the latter piece.

This is repeated for each half all the way back up to the whole, with the net effect being that the evenly indexed and oddly indexed 64 bit values are swapped such that the evens are in the first half of the bank and the odds are in the latter half of the bank.

If that's confusing, don't worry; let's break `unshuffle()` down one step at a time, using a 64-bit value array of length 16 for the sake of example. Here's what the recursive calls look like:

> Let `buf` = `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]`
> $\forall x \in buf, \text{sizeof}(x) = 64$
>
> `unshuffle(buf, 16)`
> | $len \neq 2$ and $4 | len$, so we continue...
> | `unshuffle(buf, 8)` to process the first half: `[1, 2, 3, 4, 5, 6, 7, 8]`
> | || $len \neq 2$ and $4 | len$, so we continue...
> | || `unshuffle(buf, 4)` to process the first half: `[1, 2, 3, 4]`
> | || ¦ $len \neq 2$ and $4 | len$, so we continue...
> | || ¦ `unshuffle(buf, 2)` to process the first half: `[1, 2]`
> | || ¦ | $len = 2$; return
> | || ¦ `unshuffle(buf + 2, 2)` to process the second half: `[3, 4]`
> | || ¦ | $len = 2$; return
> | || ¦ Recursion of `unshuffle(buf, 4)` over; finish call by swapping second
> | || ¦ half of first half with first half of second half
> | || **Result**: `[1, 2, 3, 4] => [1, 3, 2, 4]`
> | ||
> | || `unshuffle(buf + 4, 4)` to process the second half: `[5, 6, 7, 8]`

$len \neq 2$ and $4|len$, so we continue...
`unshuffle(buf + 4, 2)` to process the first half: `[5, 6]`
$len = 2$; return
`unshuffle(buf + 4, 2)` to process the second half: `[7, 8]`
$len = 2$; return
Recursion of `unshuffle(buf + 4, 4)` over; finish call by swapping second half of first half with first half of second half
Result: `[5, 6, 7, 8] => [5, 7, 6, 8]`

Recursion of `unshuffle(buf, 8)` over; finish call by swapping second half of first half with first half of second half
Result: `[1, 3, 2, 4, 5, 7, 6, 8] => [1, 3, 5, 7, 2, 4, 6, 8]`

`unshuffle(buf + 8, 8)` to process the second half: `[9, 10, 11, 12, 13, 14, 15, 16]`
$len \neq 2$ and $4|len$, so we continue...
`unshuffle(buf + 8, 4)` to process the first half: `[9, 10, 11, 12]`
$len \neq 2$ and $4|len$, so we continue...
`unshuffle(buf + 8, 2)` to process the first half: `[9, 10]`
$len = 2$; return
`unshuffle(buf + 8 + 2, 2)` to process the second half: `[11, 12]`
$len = 2$; return
Recursion of `unshuffle(buf + 8, 4)` over; finish call by swapping second half of first half with first half of second half
Result: `[9, 10, 11, 12] => [9, 11, 10, 12]`

`unshuffle(buf + 8 + 4, 4)` to process the second half: `[13, 14, 15, 16]`
$len \neq 2$ and $4|len$, so we continue...
`unshuffle(buf + 8 + 4, 2)` to process the first half: `[13, 14]`
$len = 2$; return
`unshuffle(buf + 8 + 4 + 2, 2)` to process the second half: `[15, 16]`
$len = 2$; return
Recursion of `unshuffle(buf + 8 + 4, 4)` over; finish call by swapping second half of first half with first half of second half
Result: `[13, 14, 15, 16] => [13, 15, 14, 16]`

Recursion of `unshuffle(buf + 8, 8)` over; finish call by swapping second half of first half with first half of second half
Result: `[9, 11, 10, 12, 13, 15, 14, 16] => [9, 11, 13, 15, 10, 12, 14, 16]`

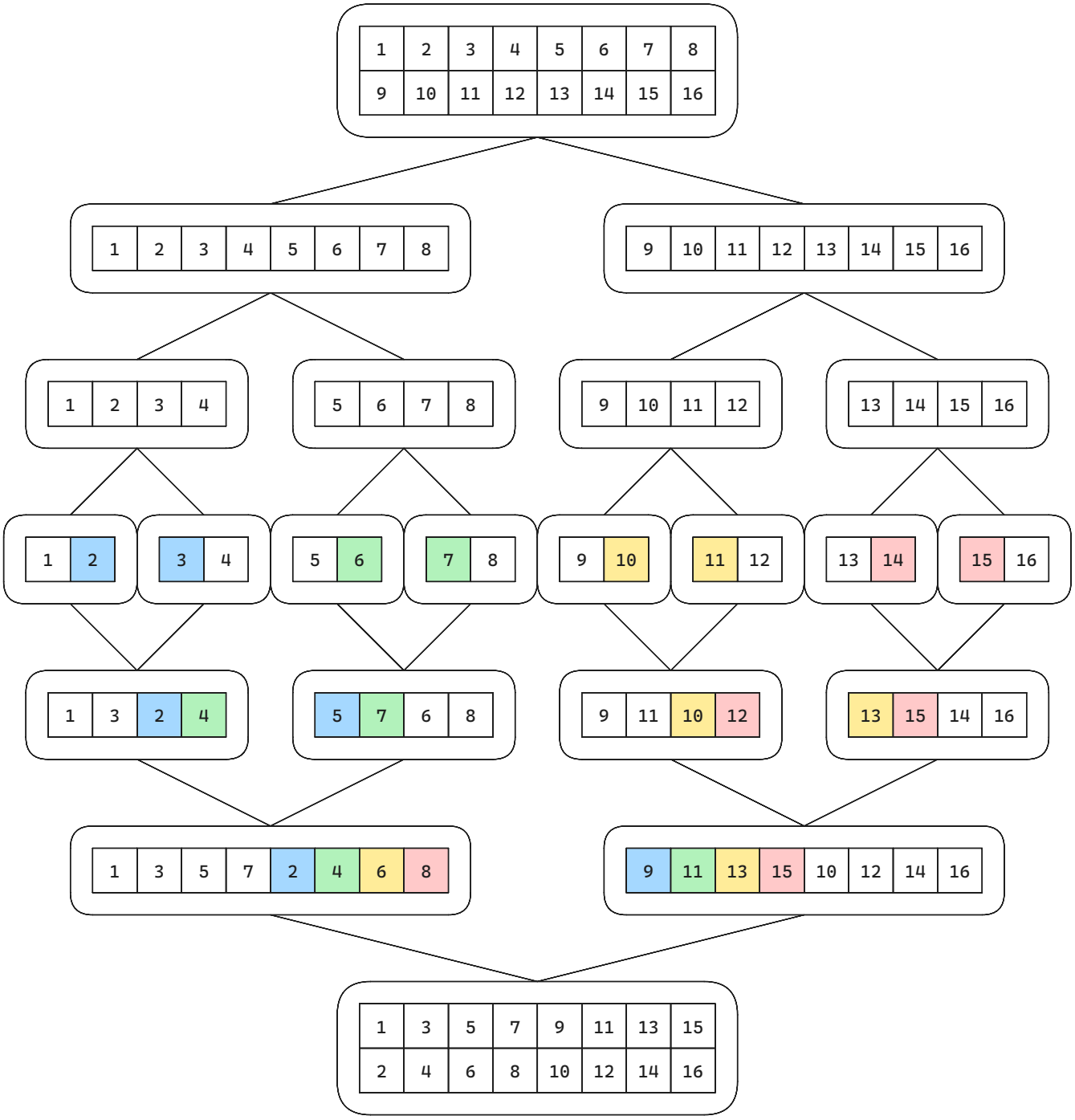
Recursion of `unshuffle(buf, 16)` over; finish call by swapping second half of first half with first half of second half
Result:

> `[1, 3, 5, 7, 2, 4, 6, 8, 9, 11, 13, 15, 10, 12, 14, 16] =>`
> `[1, 3, 5, 7, 9, 11, 13, 15, 2, 4, 6, 8, 10, 12, 14, 16]`

The final result of this algorithm is that `buf` was rearranged from the original
`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]`
to
`[1, 3, 5, 7, 9, 11, 13, 15, 2, 4, 6, 8, 10, 12, 14, 16]`

As earlier noted, values at originally at index n end up grouped and values at index $n+1$ end up grouped; that is, the oddly indexed values are grouped and the evenly indexed values are grouped.

More visually:

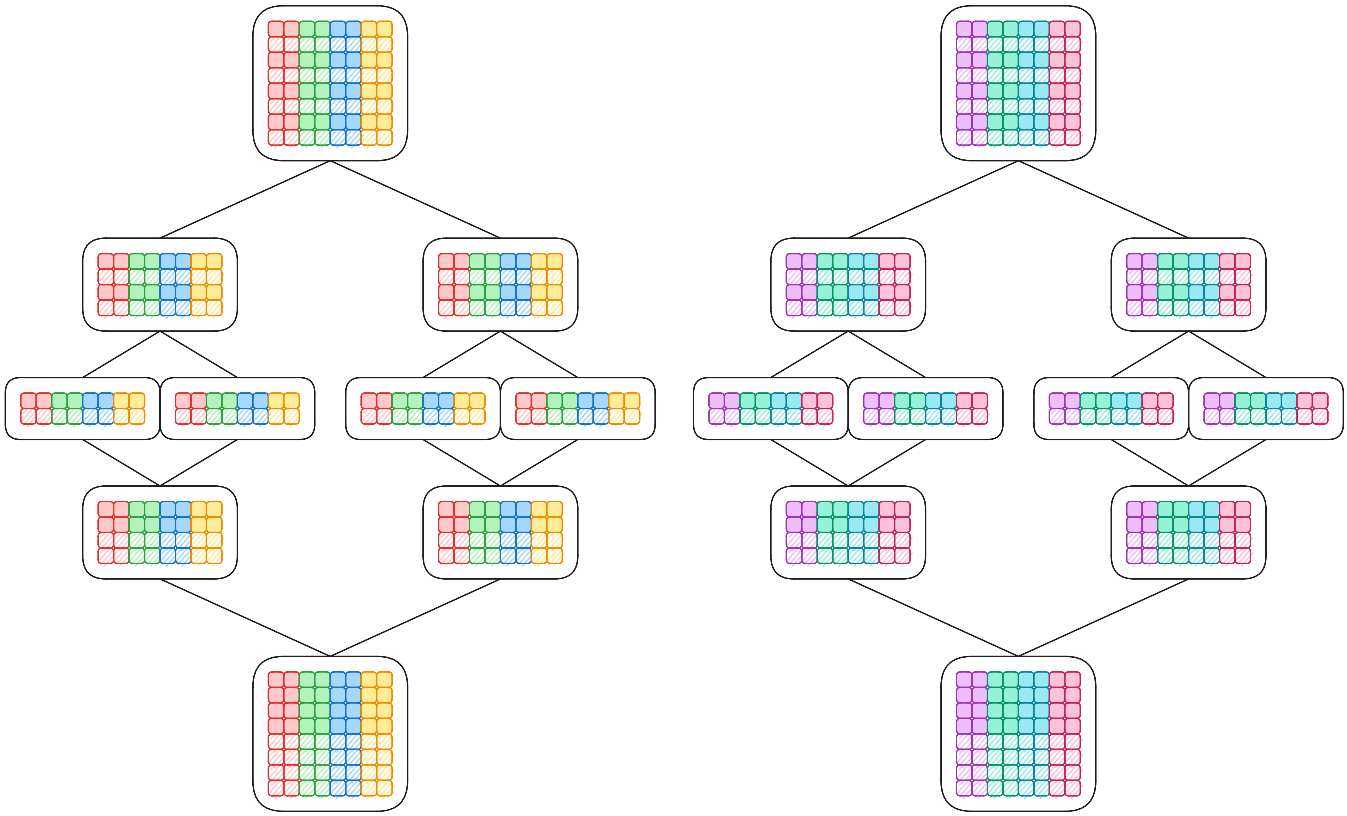


Matching colors mean those elements are swapped before the recursion layer is resolved.

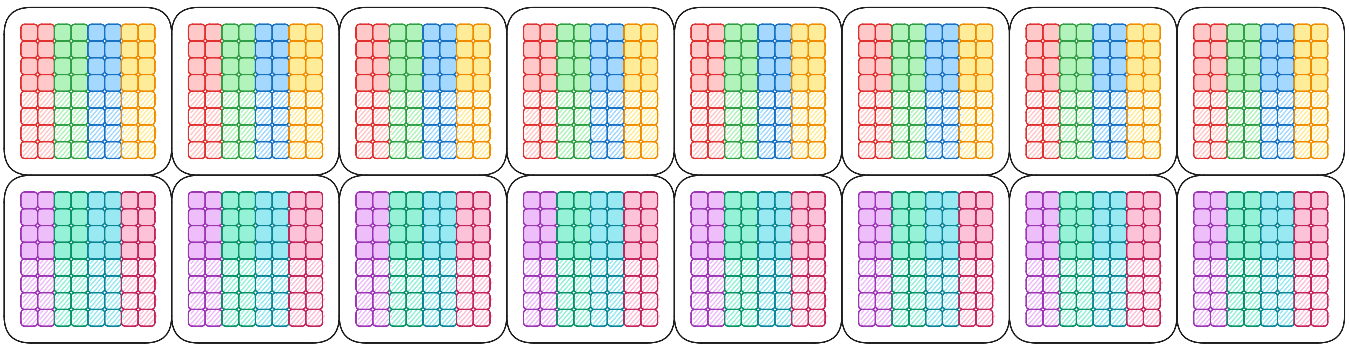
Putting It Together

Understanding how the swaps happen, we can now examine how the algorithm operates on each 2 MB bank of the interleaved files.

Each small square in the below diagram represents 1 byte, and while they only continue for 8 rows, each block represents a 2 MB bank. Filled squares represent evenly numbered 64 bit chunks while the shaded squares represent oddly numbered 64 bit chunks.



The first 8 banks are made up of the first half of the interleaved binary, which are the first 4 files. The latter 8 banks are made up of the latter half of the interleaved binary (the latter 4 files).



Stuff for nerds (algorithm time complexity analysis)

The base case, when $len = 2$, represents a constant time operation (swapping two elements), which can be considered $O(1)$.

The algorithm makes two recursive calls on arrays of size $\frac{len}{2}$ at each level of recursion, meaning the total number of recursive calls can be expressed as $T(n) = 2T(\frac{n}{2})$

Master theorem

Recall:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n),$$

where n is the size of the input problem, a is the number of subproblems in the recursion, and b is the factor by which the subproblem size is reduced in each recursive call ($b > 1$)

In the non-base case, the unshuffling operation swaps $\frac{len}{2}$ elements within the array, which can be considered $O(n)$.

Thus the recurrence relation is $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Master theorem

Recall:
If:

$$f(n) = \Theta\left(n^{\log_b(a)}\right),$$

then:

$$T(n) = \Theta\left(n^{\log_b(a)} \log n\right)$$

$f(n) = O(n)$ and $\log_b(a) = \log_2(2) = 1$, thus the time complexity of the algorithm is:

$$T(n) = \Theta\left(n^{\log_b(a)} \log n\right) = \Theta\left(n^1 \log n\right) = \Theta(n \log n)$$