A Data Structure Design for Variable-Length Coding Hardware Implementation

Zitao Fang

May 19, 2021

1 Introduction

The Lempel-Ziv compression algorithm [1] is one of the most widely used lossless compression algorithm families for generic data. Most mainstream lossless archive formats, such as ZIP, gzip, and .xz, are based on the algorithms derived from the original Lempel-Ziv algorithm LZ77 and LZ78. Beside their obvious application in data storage, these algorithms are crucial for communication and data transmission over the Internet as they reduce network congestion and transmission time.

The original algorithm is dictionary-based, involving dictionary lookup to replace repeated data with a pointer to their previous occurrences. There are already efficient hardware accelerators for this find-and-replace step, including commercial implementations such as [2] and [3] and open-source implementions. One of the open-source implementations is [4] for a simple Lempel-Ziv algorithm Snappy, which this project intends to extend for acceleration of other compression algorithms. However, most modern variants keep this step and use entropy encoding to compress the output of dictionary lookup, which would likely dominate the run time after the dictionary lookup accelerator is applied.

Huffman coding is one of the most widely used entropy encoding methods for compression algorithms. It is used by the popular general-purpose compression algorithm DEFLATE (implemented by GZIP and ZIP) [5] and many domain-specific compression algorithms, like JPEG for still images [6].

This paper introduces an encoder and a decoder for Huffman coding and any variable-length coding in general. We design the encoder and the decoder as one co-processor, running parallel to the CPU and other components of the accelerator for the entire compression algorithm. The structure of the lookup tables used by the encoder and the decoder are the same, enabling us to share the hardware LUT resources and reducing the computation overhead for generating different tables for encoding and decoding.

2 Background and Prior Works

2.1 Huffman Coding and Variable-Length Coding

Huffman code is a popular type of prefix code used for data compression. Given a list of symbols and the probability they will appear in some data flow (often approximated with its frequency in a sample dataset), we create a list of binary tree leaf nodes for each of them. Then, for the two nodes with the lowest frequencies, we create a common parent node for them, set its frequency to the sum of the child node probabilities, and replace the two nodes in the list with this single parent node. We keep doing this until there is only one root node remaining in the list, and it is the root of the so-called Huffman tree [7] .

Huffman tree is one type of variable-length code tree used for data compression. Not all variable-length code trees are Huffman trees: sometimes, the frequency used to build the tree doesn't match the symbol frequency of the input data. For example, we use predefined code trees from the algorithm specification in some compression algorithms like MPEG-4 for video encoding [8].

To encode a symbol, we will search for that symbol in the variable-length code tree from the root. For each step we move, we output 0 if we go left and 1 if we go right. The variable-length code representation for the symbol is the bit string output when we travel from the root to the corresponding leaf node.

2.2 Variable-length Coding in Hardware

Due to its importance in audio and video application, there are many hardware encoding and decoding implementation for variable-length coding. The naive approach of encoding and decoding is to walk the tree from the root, one bit (one edge) at a step until we reach the target leaf node with the desired symbol. This method is slow and inefficient in terms of its utilization of hardware resources.

[9] proposed a pointer-based encoding and decoding method. This method can have very small lookup table storage requirements compared to other algorithms, but this comes with limited throughput. This method organizes the Huffman tree as a 2D array of states, where the edges marked with 1 are vertical edges and edges with 0 are horizontal edges. The positions of the symbols in this table encode their corresponding variable-length codes. Encoding and decoding with this table can be very slow as we must access the memory once for every bit in the coded bitstream.

A variant described in [10] uses a section-based method. It groups the symbol with some common fixed-length prefixes, like 2 and 4 bits. In the paper's example, 2-bit prefixes are for the symbols with 2 and 3 bits codeword lengths, and 4-bit prefixes are reserved for longer code lengths. This structure improves the performance for short codewords as we can encode and decode multiple bits in the same cycle. There is a trade-off between the number of entries required and the decoding latency for the remaining bits after the common prefix: more granular prefix length increases the number of sections but decreases the number of memory accesses (and therefore clock cycles) for the remaining bits.

There are some partition-based decoding-only methods like [11] and [12]. The one demonstrated in [12] partition the Huffman tree based on some common prefix similar to [10]. However, this method store the remaining bits (usually very short, within 4 bits) as table entry along with code length information. [11] partitions the Huffman tree horizontally, such that every partition is a 5-level tall tree. For each partition, there will be a table storing the encoding information for its nodes. This (variable-length) table includes the remaining code length and the next memory location to look if the node is not a leaf. These methods are decoding only and could not be used for encoding.

Another decoding-only method described in [13] uses a shift register to store the code bitstream that progresses one bit per cycle. As we read more bits in and traversing down the code tree, the algorithm narrow down the range of possible encoding that corresponds to the target symbol leaf. This method is lightweight, having small combinational logic, but its decoding speed is limited to one code bit per cycle.

3 Proposed Data Structure and Algorithm

3.1 LUT Data Structure

We propose a lookup table structure and corresponding encoding/decoding algorithm for a 256-symbol alphabet (8-bit symbol) in this paper. In this design, we store the radix sort orders of the symbols in the Huffman tree in our table. The radix sort order is the order of a symbol after radix sort on the binary string of their Huffman code. In a Huffman tree, the symbols with lower radix sort order will always be on the left. To convert between symbols and orders, two 256-entry tables with 8-bit entries are required.

To compactly store a Huffman tree without negatively affecting the throughput, we split the entire tree into a series of four-level subtrees, similar to the method in [11]. Every leaf node in the subtree represents a 4-bit substring of some symbols' Huffman codes.

We will store the information of a subtree in two 16-entry tables, where every table entry corresponds to one of the nodes in this tree. In the first table, the entry will store the maximal radix sort order of the symbol under the corresponding subtree leaf node, and we will call this table the max order table (`maxtable`). The second table will store the index of the next 5-level subtree under this node, which we call the next subtree table (`nexttable`). All entries are 8-bit wide.

For a true leaf node within the subtree, we assume that there are some imaginary subtree leaves under it as if the leaf node were an internal node in a full 5-level binary tree. We set the max order entry under this node to be the radix order of the symbol it represents, and we set the corresponding next subtree entry to 0. Subtree 0 always represents the 5-level subtree at the Huffman tree root, and any zero value entry in the next subtree table implies that we have reached a leaf node and will go back to the root node next.

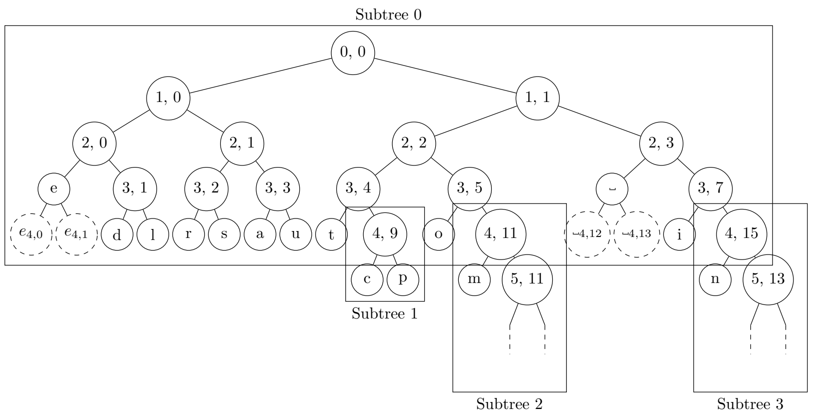

Figure 1: Part of a Huffman tree for *Lorem Ipsum*, divided into subtrees.

Table 1: Radix sort order of selected symbols.

Symbol	Order
e	0
d	1
l	2
r	3
s	4
a	5
u	6
t	7
o	10
m	11
(whitespace)	16
i	17

Fig. 1 is an example Huffman tree generated from our test dataset, *Lorem Ipsum* [14]. We only show a part of the tree for simplicity. Any nodes in dashed circles are the "imaginary" nodes used to fill in the 5-level subtree (like the nodes below e and ␣), so that we can create standard size tables for every 5-level subtree. These imaginary nodes has the same property as the actual leaf node in the tree. Only the 16 nodes on the bottom of a subtree will have corresponding entries in the table.

Table 1 is a table of raidx sorted order for the symbol in Subtree 1. It is obvious that the symbol on the left of node 5, 11 (where we start to have deeper tree omitted from the graph) has their radix sort order equal to the order of leaf from the left. The generated `maxtable` and `nexttable` for the subtree 0 in Fig. 1 is shown in Table 2.

We haven't deduced the maximal number of subtrees for a 256-symbol Huff-

Table 2: Generated lookup table (Subtree 0).

Index	`maxtable`	`nexttable`
0	0	0
1	0	0
2	1	0
3	2	0
4	3	0
5	4	0
6	5	0
7	6	0
8	7	0
9	9	1
10	10	0
11	15	2
12	16	0
13	16	0
14	17	0
15	255	3

man tree, but we believe that 64 subtrees are sufficient to cover the entire tree.

Usually, the table is generated by software, although hardware can also generate the Huffman tree and construct the table. Here is the software algorithm to generate the table, assuming that we already have a binary tree structure:

1. Do an in-order traversal to the Huffman tree. For every leaf node we encounter, we replace the symbol stored in the leaf with the leaf's order of visit (equivalent to its radix sort order). Also, we save the symbol and the leaf's order to two lookup tables so that we can convert between them.

2. Define a recursive process that will return the maximal order for the 5-level subtree it processes and its index:

   - Allocate a subtree index for the current block.
   - Do a breadth-first search starting at the subtree root, with a maximum distance of 4. Save the node we have in the search queue and any leaf nodes encountered into a list, in the order from left to right.
   - For an internal node that is four edges away from the current subtree root node, run the recursive process on the node. Write the maximum order and the index in the `nexttable` to the table. For a leaf node, fill the next $2^{4-\text{level}}$ `maxtable` entry with the radix sort order of the leaf, then fill the corresponding `nexttable` entry with 0.
   - Return the 15th entry of the current `maxtable` and allocated subtree index.

3. Execute the process defined in Step 2 on the root of the Huffman tree.

The pseudocode of this algorithm is listed below as Algorithm 1.

Algorithm 1 Generate table contents

Require: Spaces are allocated for table $maxTable, nextTable, sym2Ord, ord2Sym$

```
$next\_subtree\_idx \leftarrow 0$
$current\_order \leftarrow 0$
function BuildTable($node$)
    $leaves \leftarrow$ BreadthFirstSearch($node$,4)   ▷ It return a list of 4th-order children
    $i \leftarrow 0$
    $current\_table \leftarrow next\_subtree\_idx$
    $next\_subtree\_idx \leftarrow next\_subtree\_idx + 1$
    for $leaf \leftarrow leaves$ do
        if $!leaf.huffman\_leaf$ then   ▷ Check if this is an actual leaf
            $current\_max, next\_table \leftarrow$ BuildTable($leaf$)
        else
            $next\_table \leftarrow 0$   ▷ 0 represents leaf node
            $current\_max \leftarrow current\_order$
            $sym2Ord[leaf.symbol] \leftarrow current\_order$
            $ord2Sym[current\_order] \leftarrow leaf.symbol$   ▷ Record order entry
            $current\_order \leftarrow current\_order + 1$   ▷ Update order
        end if
        for $d \leftarrow 0 \rightarrow 2^{leaf.dist}$ do   ▷ Write table entry
            $maxTable[current\_table][i] \leftarrow current\_max$
            $nextTable[current\_table][i] \leftarrow next\_table$
            $i \leftarrow i + 1$
        end for
    end for
    return $current\_max, next\_subtree\_idx$
end function
```

The algorithm described above is just an example, and there may be multiple ways to generate the table. Our software library uses a slightly different method to generate tables from the source Huffman tree.

3.2 Encoding Algorithm

The encoding algorithm can be described as follows:

1. Save the starting address and the length of the symbol stream. Set the next symbol to the one located at the starting address.
2. Move to the next symbol, and query the symbol-order table to translate the symbol into the radix sort order. If there are no more input symbol,

terminate the loop.

3. Read all 16 entries from the current `maxtable`. This requires two 64-bit port reading simultaneously from a RAM port.

4. Find the lowest entry from the `maxtable` that is greater or equal to the symbol order. Save its index.

5. Read the current next subtree table, with the index equal to that from Step 4. Set the current subtree index for next cycle to the read value.

6. If the value from Step 5 is equal to 0, find the indices of the highest entry and the lowest entry equal to the output symbol, output their common prefix, and go back to Step 2. Otherwise, output the value from step 4, and go back to Step 3.

The pseudocode of this algorithm is listed below as Algorithm 2.

Algorithm 2 Encoding Algorithm

Require: Table $maxTable, nextTable, sym2Ord$ are defined, input symbols stored in mem

```
function ENCODE($addr, len$)
    $current\_table \leftarrow 0$
    for $i \leftarrow 0, len - 1$ do
        $order \leftarrow sym2Ord[mem[addr + i]]$
        $is\_leaf \leftarrow$ False
        while $\neg is\_leaf$ do
            for $j \leftarrow 0, 15$ do
                $eq[j] \leftarrow maxTable[current\_table][j] = order$
                $lt[j] \leftarrow maxTable[current\_table][j] < order$
            end for
            $match\_low \leftarrow$ LOWESTTRUEINDEX($lt$)
            $next\_table \leftarrow nextTable[current\_table][match\_low]$
            $is\_leaf \leftarrow next\_table = 0$
            $shamt \leftarrow 4 - \log_2($POPCOUNT$(lt))$
            BITSTREAMOUT($match\_low, shamt$)
            $current\_table \leftarrow next\_table$
        end while
    end for
end function
```

The basic idea of this algorithm is to locate the correct subtree that our symbol is at and output the code (which often equal to the index of the entry in max table) along the way. Remember that we are comparing the radix sort order of the symbol, which can only increase as we go to the right (or higher index in a table). Assume the entry index returned by Step 3 is i, we are sure that any symbols under the $(i-1)$-th node will always have a smaller order than the current symbol, and all symbols under $(i+1)$-th node will have a greater order. Therefore, our input symbol must be under (or at) the node returned by Step 3.

Even if we are already at the subtree block where our symbols locate, the desired leaf node can be at any level. We can find the correct level (and therefore the correct number of bits to output) by looking at how many entries in the `maxtable` are having the same value and equal to the radix sort order of our symbol. If the symbol is at level 1 (level 0 is the subtree root), we will have 8 entries with the same value where their index share the highest bit. We can therefore simply output the common prefix to our Huffman code stream, which is the highest bit here. Similarly, there will be 4 common entries for level 2 leaf, 2 entries for level 3, and 1 entry for level 4.

In hardware implementation, Step 1 can often be pipelined, feeding the result to the main loop from Step 2 to Step 5 once the current subtree index is going back to 0. In addition, the indices in Step 3 and Step 5 can be generated in the same cycle and decrease the latency. In our implementation, the main loop from 2 to 5 only takes 2 clock cycles.

We use a shift buffer to store the output of this algorithm. The shift buffer has 12 bits, and it will output the highest valid 8 bits once there are more than 8 bits present in the buffer. Note that we will only output string with length 1 to 4 every clock cycle, and 12 bits is sufficient to handle them.

3.3 Decoding Algorithm

The encoding algorithm can be described as follows:

1. Move to the next symbol, and query the symbol-order table to translate it into the radix sort order. If there are no more input symbols, terminate the loop.

2. Read all 16 entries from the current `maxtable`.

3. Find the lowest entry from the `maxtable` that is greater or equal to the symbol order. Save its index.

4. Read the current next subtree table, with the index equal to that from Step 3. Set the current subtree index for the next cycle to the return value of the read.

5. If the value from Step 4 is equal to 0, find the indices of the highest entry and the lowest entry equal to the output symbol, output their common prefix, and go back to Step 1. Otherwise, output the value from step 3, and go back to Step 2.

The pseudocode of this algorithm is listed below as Algorithm 3.

The basic idea of this algorithm is to locate the correct subtree that our symbol is at and output the code (which often equal to the index of the entry in

Algorithm 3 Decoding Algorithm

Require: Table $maxTable, nextTable, ord2Sym$ are defined, a bitstream is reading from the memory for codes

```
function DECODE($addr, len$)
    $current\_table \leftarrow 0$
    for $i \leftarrow 0, len - 1$ do
        $is\_leaf \leftarrow$ False
        while $\neg is\_leaf$ do
            $index \leftarrow$ PEEKBITSTREAM(4)
            $next\_table \leftarrow nextTable[current\_table][index]$
            if $next\_table = 0$ then
                $is\_leaf \leftarrow$ True
                $order \leftarrow maxTable[current\_table][index]$
                OUTPUTSYMBOL($ord2Sym[order]$)
                for $j \leftarrow 0, 15$ do
                    $eq[j] \leftarrow maxSymbol[current\_table][j] = order$
                end for
                $shamt \leftarrow 4 - \log_2(\text{POPCOUNT}(eq))$
                SHIFTBITSTREAM($shamt$)
            end if
            $current\_table \leftarrow next\_table$
        end while
    end for
end function
```

the `maxtable`) along the way. Remember that we are comparing the radix sort order of the symbol, which can only increase as we go to the right (or higher index in a table). Assume the entry index returned by Step 3 is i, we are sure that any symbols under the $(i-1)$-th node will always have a smaller order than the current symbol, and all symbols under $(i+1)$-th node will have a greater order. Therefore, our input symbol must be under (or at) the node returned by Step 3.

Even if we are already at the subtree block where our symbol locates, the desired leaf node can be at any level. We can find the correct level (and therefore the correct number of bits to output) by looking at how many entries in the `maxtable` are having the same value and equal to the radix sort order of our symbol. If the symbol is at level 1 (level 0 is the subtree root), we will have 8 entries with the same value where their index share the highest bit. We can therefore simply output the common prefix to our Huffman code stream, which is the highest bit here. Similarly, there will be 4 common entries for level 2 leaf, 2 entries for level 3, and 1 entry for level 4.

In a hardware implementation, this algorithm requires two independent 64-bit read ports. To improve the throughput, we can pipeline Step 1 and feed the result into the main loop from Step 2 to Step 5 when the current subtree index is about to count to 0. This can reduce the clock cycles per iteration to 2. Merging the pipeline step further will increase the path delay significantly in our implementation.

We use a shift buffer to store the output of this algorithm. The shift buffer has 12 bits, and it will output the highest valid 8 bits once there are more than 8 bits present in the buffer. Note that we will only output string with length 1 to 4 every clock cycle, and 12 bits is sufficient to handle them.

4 Implementation and Result

For implementation, we use the architecture shown in 2.

We implemented the algorithm described in the previous section with ASAP7 7nm technology, an open-source predictive PDK. Due to the lack of an open-source SRAM compiler for this technology, we used predictive models for the LUT RAM instead of the actual SRAM block for this project. The accelerator is implemented using Chisel language, a hardware language developed at UC Berkeley [15]. The Chisel compiler generates synthesizable Verilog code, and we run the synthesis and place-and-route workflow with the Hammer framework, a part of UC Berkeley's Chipyard SoC development framework [16]. All code required to reproduce the result is provided in the accompanying open-source repository.

For performance testing, we used a 1000-character long sample text (the "Lorem Ipsum" text) as our input and build a Huffman tree based on the symbol frequency in the text [14]. The optimal Huffman code for this text produces 523 bytes of output. We measure the cycles by running unit tests on the original Chisel code before transformation using the Treadle test tools. With some test

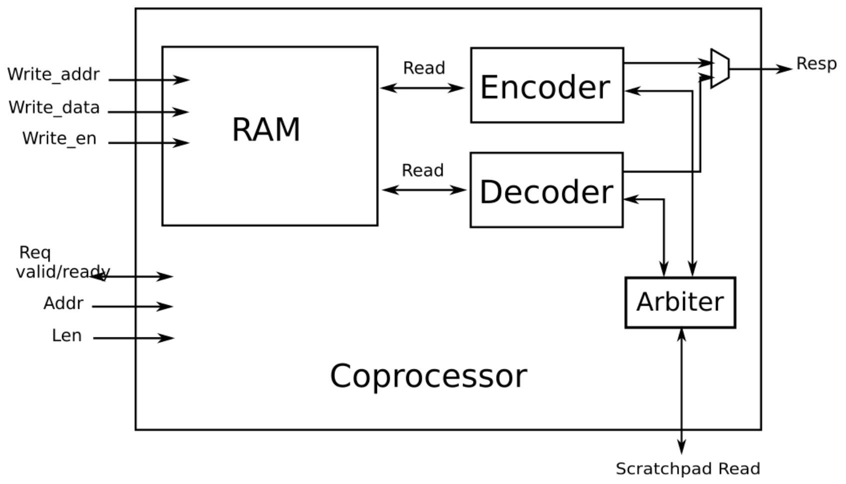

Figure 2: Accelerator structure.

overhead, the encoding takes 2482 cycles to finish, and decoding takes 1246 cycles. The functional verification of the implementation is done with Verilog testbench using the same dataset as performance testing on Chisel level. The Verilog testbench produces the same inputs and expects the same outputs from the Treadle testbench, and a functionally correct implementation should have identical behavior as the Chisel RTL model and pass all tests in the Verilog testbench.

The implementation is placed in a 190μm by 260μm region. We can achieve a 1GHz operating frequency with this technology. After place and route, the total area of encoder, decoder, and SRAM is 31604 μm^2. Encoder alone takes 2754 μm^2, and decoder takes 2306 μm^2. The RAM consists of four 128x32 dual-port SRAM blocks and two 64x32 single-port SRAM blocks. Fig. 3 and 4 show the Ameoba view and the Physical view of the final layout.

5 Conclusion

Because of the wide adoption of variable-length coding in data compression, a more efficient encoding/decoding algorithm can often greatly benefit storage, multimedia, and network applications. Our algorithm reduces the table storage requirement for encoding and decoding while maintaining the speed, and it is best suited for the high-performance application that performs both encoding and decoding operations, like distributed data processing nodes, workstations for video rendering, etc. In the future, we plan to integrate this implementation of variable-length coding into accelerators that target the entire compression algorithm like DEFLATE.

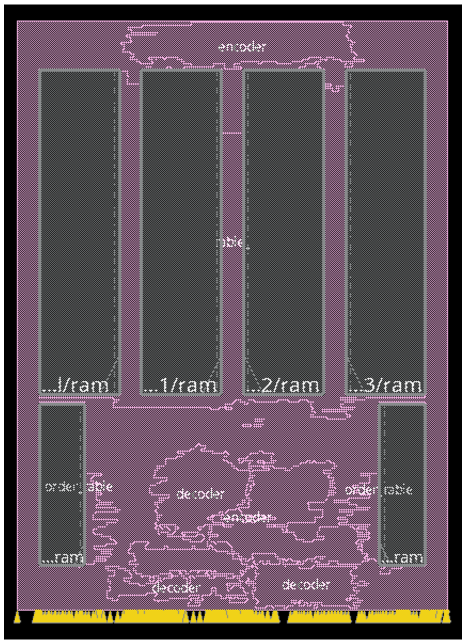

Figure 3: Ameoba view of the final layout.

References

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE TRANSACTIONS ON INFORMATION THEORY*, vol. 23, no. 3, pp. 337–343, 1977.

[2] C. F. Webb, "Ibm z10: The next-generation mainframe microprocessor," *IEEE micro*, vol. 28, no. 2, pp. 19–29, 2008.

[3] X. Hu, F. Wang, W. Li, J. Li, and H. Guan, "QZFS: QAT accelerated compression in file system for application agnostic and cost efficient data storage," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 163–176, ISBN: 978-1-939133-03-8. [Online]. Available: `https://www.usenix.org/conference/atc19/presentation/hu-xiaokang`.

[4] K. Kovacs, "A hardware implementation of the snappy compression algorithm," *UC Berkeley EECS Masters Thesis*, 2019.

[5] P. Deutsch, *Rfc1951: Deflate compressed data format specification version 1.3*, 1996.

[6] G. K. Wallace, "The jpeg still picture compression standard," *IEEE transactions on consumer electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

[7] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

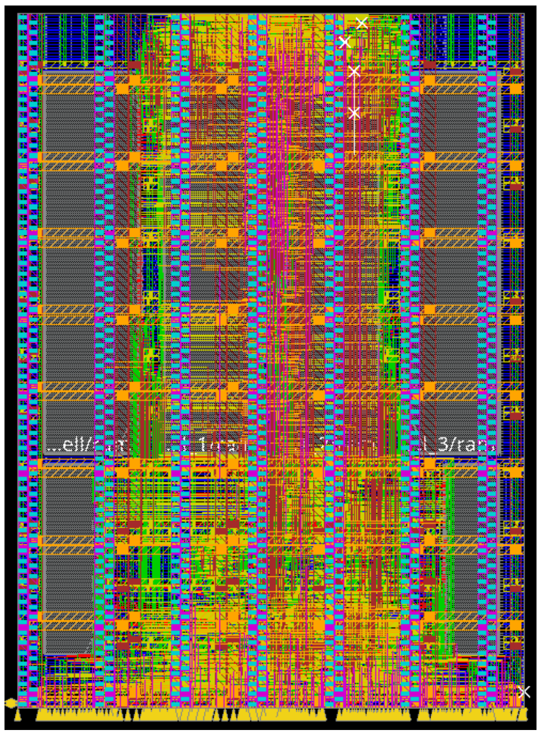
Figure 4: Physical view of the final layout.

[8] D. Salomon, *Variable-length codes for data compression.* Springer Science & Business Media, 2007.

[9] G. Higgie and A. Fong, "Efficient encoding and decoding algorithms for variable-length entropy codes," *IEE Proceedings-Communications*, vol. 150, no. 5, pp. 305–311, 2003.

[10] Y. Lin and P.-Y. Chen, "A low-cost vlsi implementation for vlc," in *2006 1ST IEEE Conference on Industrial Electronics and Applications*, IEEE, 2006, pp. 1–4.

[11] R. Hashemian, "Design and hardware implementation of a memory efficient huffman decoding," *IEEE Transactions on Consumer Electronics*, vol. 40, no. 3, pp. 345–352, 1994.

[12] S. B. Choi and M. H. Lee, "High speed pattern matching for a fast huffman decoder," *IEEE Transactions on Consumer Electronics*, vol. 41, no. 1, pp. 97–103, 1995.

[13] B. Nikolic, B. Wild, V. Dai, *et al.*, "Layout decompression chip for maskless lithography," in *Emerging Lithographic Technologies VIII*, International Society for Optics and Photonics, vol. 5374, 2004, pp. 1092–1099.

[14] Letraset. "Lorem ipsum." (1950), [Online]. Available: `https://wikisource.org/wiki/Lorem_ipsum` (visited on 05/08/2021).

[15] J. Bachrach, H. Vo, B. Richards, *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, IEEE, 2012, pp. 1212–1221.

[16] A. Amid, D. Biancolin, A. Gonzalez, *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.