Using Dynamic Shifters for Linear Address Bitmask Generator via Chisel3

Linyuan Huang
University Of Miami, USA

ABSTRACT

Chisel is a hardware construction language that supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages. Chisel can generate a high-speed C++ based cycle-accurate software simulator, or low-level Verilog designed to a standard ASIC flow for synthesis [1]. In this project, we choose Chisel to design dynamic shifters for linear address generator bitmask generation. When accessing a banked memory, bitmask should be generated to select different bytes in one memory row across different banks. We design a linear address bitmask hardware generator by Chisel to produce programmable linear address bitmask hardware. Our experiments are based on dynamic shifters with comparison of multiplexer-based design. Based on logic synthesis results, we have achieved lower power consumption and lower area in different configurations with little frequency loss. This paper shows an alternative design for linear address bitmask generators that demonstrate possible tradeoff for PPA (performance, power and area).

CCS CONCEPTS

• **Hardware** → Integrated circuits; Logic circuits; Asynchronous circuits.

KEYWORDS

Chisel, Dynamic Shifters, Linear Address, Bit mask, Memory, Byte

ACM Reference Format:
Linyuan Huang. 2021. Using Dynamic Shifters for Linear Address Bitmask Generator via Chisel3. In *2021 4th International Conference on Electronics and Electrical Engineering Technology (EEET 2021), December 03–05, 2021, Nanjing, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3508297.3508326

1 INTRODUCTION

Scala is a general-purpose functional programming language that provides both object-oriented programming paradigm and functional programming interface. Chisel (Constructing Hardware In a Scala Embedded Language), developed by University of California, Berkeley, is an open-source hardware description language (HDL) that can be used to describe digital and analog circuits at gates/register-transfer level. As Chisel's name implies, it is developed based on Scala programming language, which means Chisel is essentially a package with s set of Scala libraries that define new hardware datatypes and a set of routines to convert a hardware data structure into a fast C++ simulator or low-level Verilog (or other RTL) for emulation or synthesis. Reason of choosing Scala are explained by Chisel developer: 1) Scala is a powerful functional programming language for building circuit generators; 2) Scala is developed as a base for domain-specific language (DSL); 3)Scala compiles to the Java Virtual Machine (JVM); 4) Scala has a large set of development support tools like IDE [2]; 5) Scala is good at trait mixin and has many features that are helpful for hardware construction. The reason for us to choose Chisel for this paper is: 1) Chisel allows users to create reusable objects and functions; 2) Chisel allows users to create and define their own data types; 3) Chisel allows users to better capture particular design pattern by writing their own data domain-specific languages on top of Chisel. Compared with a hardware implementation with Chisel and traditional Verilog implementation, the main advantage of using Chisel is that Chisel supports highly parameterized circuit generators. Based on the experiments that were done by Chisel's developers, a simple 3-stage 32-bit RISC processor [3], Chisel produced Verilog code that has 3x reduction lines of code than the conventional hand-written Verilog code. They also did some experiments to show Chisel generated Verilog code has similar area and approximately 8x speedup than hand-coded Verilog. Most modern memory designs have banks within each memory line. Accessing banked memory requires bitmasks to select from different bytes of one memory line across different banks. Thus, it is important to design small and power efficient bitmask generation hardware. We used Chisel to form a configurable generator for building a linear address bitmask generation hardware, which could be included into a configurable memory generator. Linear address bitmask generation is to generate a bitmask for accessing patterns of one line of memory. The number of bits in bitmask equals the memory width; each 1 in bitmask corresponds to each byte that needs to be accessed within the memory line, while each 0 in bitmask corresponds to bytes that are not accessed. For bitmask generation, the inputs are the line offset into each memory line, the length of the unit that needs to be accessed, the exponent of bytes that each unit is, and the direction of each access. There exist three possible directions for access: 1) ascending access (direction = 1) is accessing the byte and bytes after the first element. 2) descending access (direction = -1) is accessing the byte and bytes before the first element. 3) constant access (direction = 0) is accessing the first element only.

EEET 2021, December 03–05, 2021, Nanjing, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8516-9/21/12…$15.00
https://doi.org/10.1145/3508297.3508326

2 RELATED WORK

For a linear address bitmask hardware generator can be used for generating a sequence of addresses for the purpose of speeding up the memory access. In the paper, we mainly design a linear address

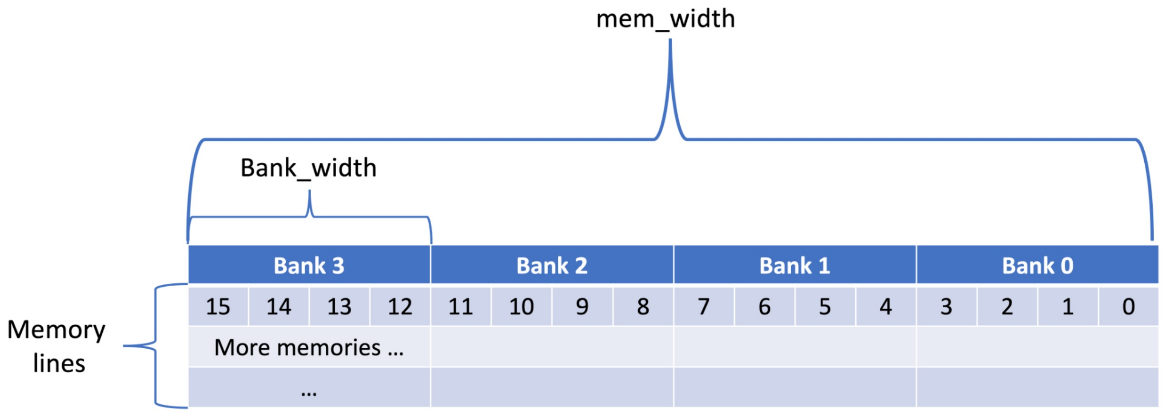

Figure 1: A 4-way banked memory with bank width of 4 bytes

```
For each possible exponent to generate a local_bitmask array and next_length array{
    Calculate number of bits in local_bitmask = mem_width << exp
    Compute range for accessing based on direction
    next_length_exp[exponent] is generated based on range and direction and exponent
    For each bit in local_bitmask{
        If the bit is in the range that need to be accessed
            Set the bit to 1
        Else
            Set the bit to 0
        }
}
global_mask = local_bitmask[exponent].expand()      // use expand() to change local to global
next_length = next_length_exp[exponent]
```

Figure 2: Baseline Algorithm for Local Mask Generation

generator for a bank-scratchpad memory like [4]. The scratchpad memory is divided into multiple lines, and each line of memory is separated by different banks. In this way, each access to memory can be separated by access to different banks, thus reducing the access time to memory and bus contention. The number of bytes in each line is called memory width, and the number of bytes in each line of each bank is called bank width. For example, suppose we have a 4-way banked memory that is byte-addressable with bank width of 4 bytes (or memory width of 16 bytes). As shown on Figure 1, for each memory line that has 16 bytes, 4 bytes are stored in each bank, and accessing the memory should cycle through bank0 to bank3.

One realization of linear address bitmask generator is to compute a local that is a reduced bit mask according to exponent. A conversion from local mask to output bitmask by duplicating bits. The abstracted algorithm for generating a local mask is described below. To be specific, if exponent = 2 (each access unit should be 2 = 4 byte), considering the example of 4-way banked byte addressable memory with memory width of 16 bytes, the memory line bitmask should be 16 bits, but the reduced local mask should be only 4 bits (calculated as 16−byte/4−byte). The local bitmask is generated bit by bit, and each bit is generated as 1 if the bit is in range of memory access request. At the final stage, the local bitmask will be converted back to global bitmask to match the width of memory, representing the whole line, and the next row number is computed.

The downside of this algorithm (figure 2) is that generating a local bitmask for every possible exponent value given memory width will result in a huge hardware overhead during logical synthesis. Generating a combinational logic for every exponent and uses the exponent to select from a list of local masks will eventually result in a large multiplexer and duplicated logics. We proposed a new

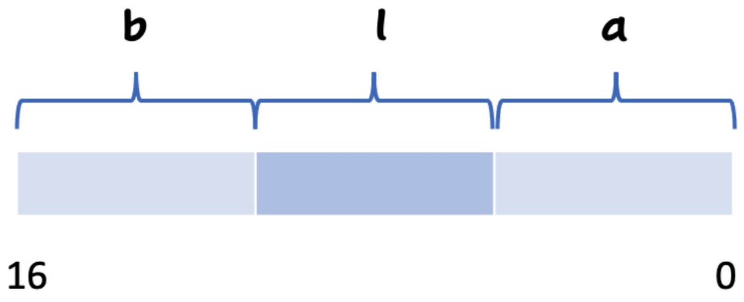

Figure 3: a, l, b for Memory Width 16

algorithm to reduce the area and power by applying a non-recursive logic, which will be explained in the next section.

3 DYNAMIC SHIFTER ARCHITECTURE

Because the exponents are determined only at runtime, only synthesizing logics targeting specific exponent values would be detrimental and would result in wrong bitmask generated. It seems inevitable that we need to form combinational logic to compute masks for each exponent. However, our new algorithm bypasses this constraint and uses dynamic shifters and adders to complete the seemingly impossible task.

3.1 Pre-computation

All bitmasks can be formed by shifting all “1” bit streams right and then left. Since we will be using shifters to form bitmasks, the first step of our method is to find out the amount to be shifted for each bitmask. We first define some variables. Suppose we have a memory with a width of 16 bytes, so we need a 16-bit bitmask. As shown in Figure 3, we define that l is the length of bits needed to be “1”, a is the length of bits before needed to be “0”, and b is the length of bits after needed to be “0”. Then the resulting bitmask will just be all “1” bit streams shift right by a + b, then shift left by a.

3.2 4 Case Studies: Bitmask Calculation under different exponents

Computation of a, b, l will be different for each direction explained by using four examples that use the same memory setup as Figure 1. Figure 4 shows the example that has the line offset = 5, length = 3, exponent = 1, direction = 1. This means that we need to access three (length = 3) two-byte (exponent = 1, 21 = 2 bytes) elements starting from the fourth byte (line offset = 5, each access should be aligned to a two-byte address) of the memory line. In other words, if the line offset is an odd number, the access should be aligned to an even number. Finally, there are 6 bytes in total that should be accessed from the fourth offset to the ninth offset. Figure 4 shows the bitmask should be generated as 0000,0011,1111,0000. In addition, the linear address bitmask generator should generate the remaining length (next length) of bytes that needed to be accessed. If we still consider the example from Figure 4, then next length = 0. From this example we can clearly see that if direction =1, then a is 4 (line offset aligned), l is 6 that is computed based on exponent and length, and b equals 6 that is computed from mem width - l - a.

Example 2 is presented in Figure 5. If we applied length = 7 and kept other parameters as same as example 2, then next length = 1. The bitmask should be 1111,1111,1111,0000, while a = 4, b = 0, and l = 12, just as last example.

Example 3 is shown in Figure 6 that is line offset = 5, length = 7, exponent = 2, direction = 0. Then we will have next length = 0 since its direction is constant. The bitmask should be 0000,0000,1111,0000, while a = 4, b = 8, and l = 4. From this example we can see that in direction = constant, l is only the length of an element, while computation for a and b stays the same.

Example 4 is shown in Figure 7 that is line offset = 5, length = 4, exponent = 2, direction = -1, other parameters keep the same as example 1. The result of bitmask is 0000,0000,1111,1111 and next length = 2. From this example we can see that in direction = descend, calculation for a, b, l is different from when direction = ascend or constant. If direction is descended, then b = mem width - line offset aligned - 1¡¡exponent, and l is determined by length and a = mem width - b - a.

As we can see from the above 4 case studies, the computation for a, b, l is straightforward for each possible direction, and would only require small adders and dynamic shifters. This would produce little area and power overhead for the whole bitmask generator. After a, b, l is acquired, we then use dynamic shifters to form bitmasks, which detailed algorithm is described below.

3.3 Dynamic Shifters Design

Dynamic shifters are common hardware components [6] that can shift a binary number based on some fixed bit width shift number. Chisel will automatically generate a dynamic shifter, if the shifting operand is a wire, which value can only be determined in runtime. As shown in Figure 8, each shifter has an enable bit that is connected to each bit of shift number. The dynamic shifter allows us to achieve left or right shifting by a dynamically determined binary number, with little area and power overhead.

To form a bitmask, we can shift an all-one-bit stream right by a+b bits and then left by a bits. For example, to form 0000,1111,0000,0000, we only need to right shift 16 bits of “1” by 12 and then left shift the result by 8. With the knowledge of dynamic shifter, we design the below algorithm. Note that next length computation would require adders and a, b, l computation would require both adders and dynamic shifters.

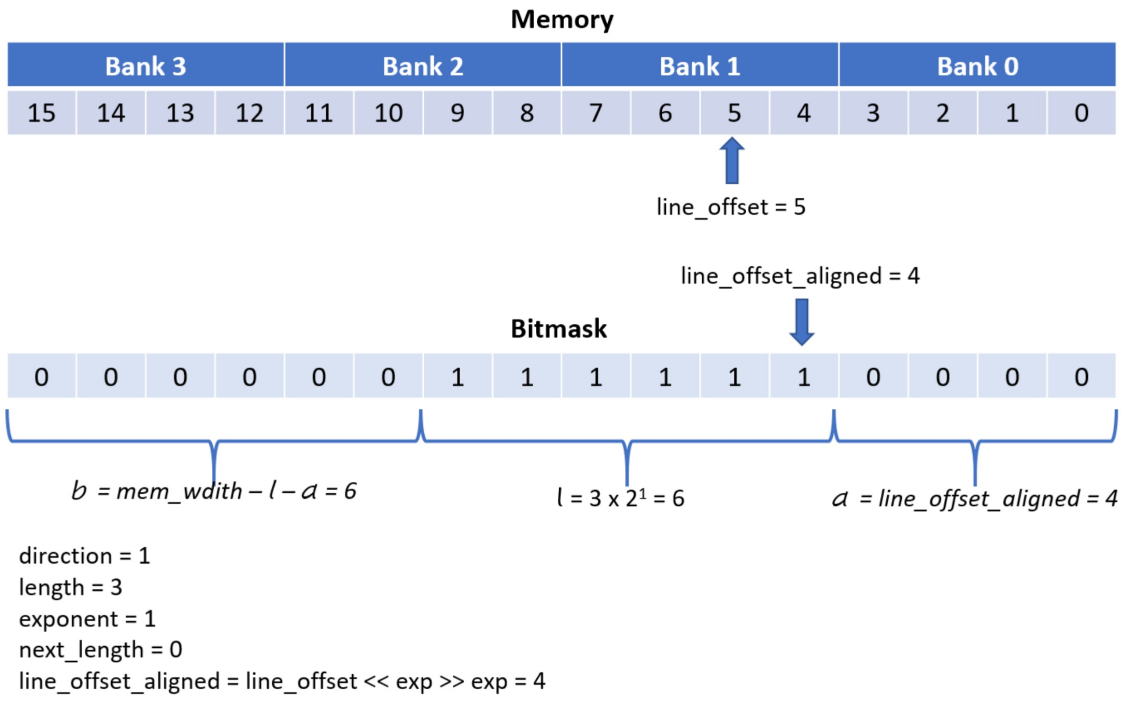

Figure 4: Example 1 shows the a, b, l computation for direction = ascend

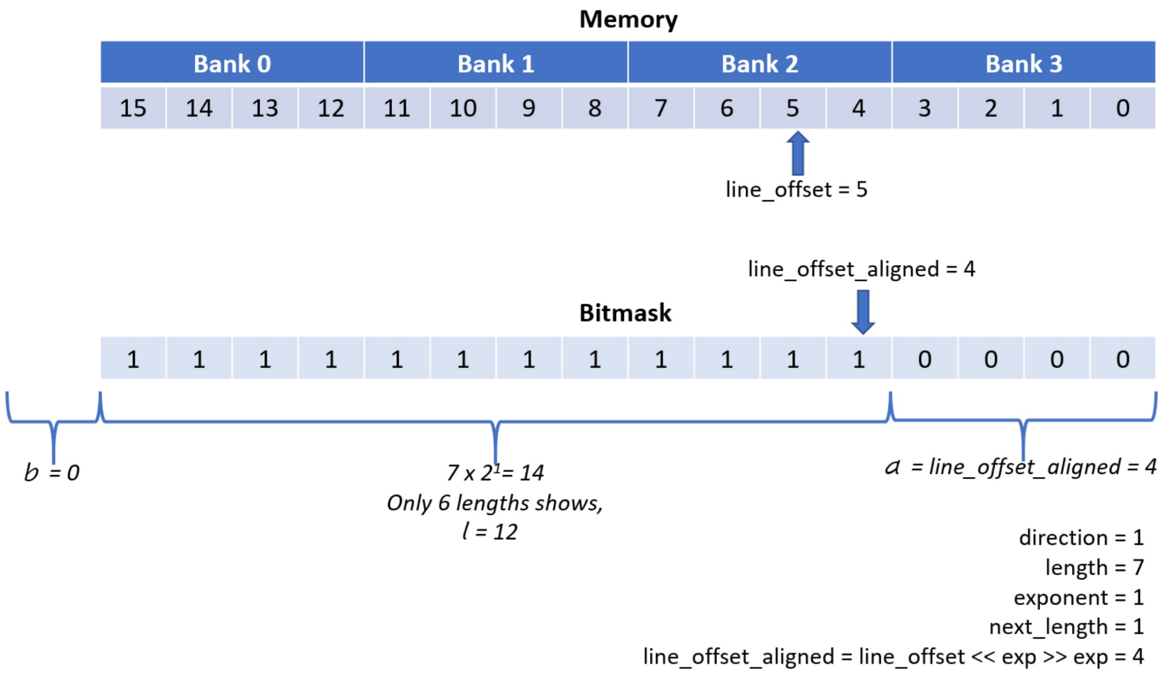

Figure 5: Example 2 shows the a, b, l computation for direction = ascend

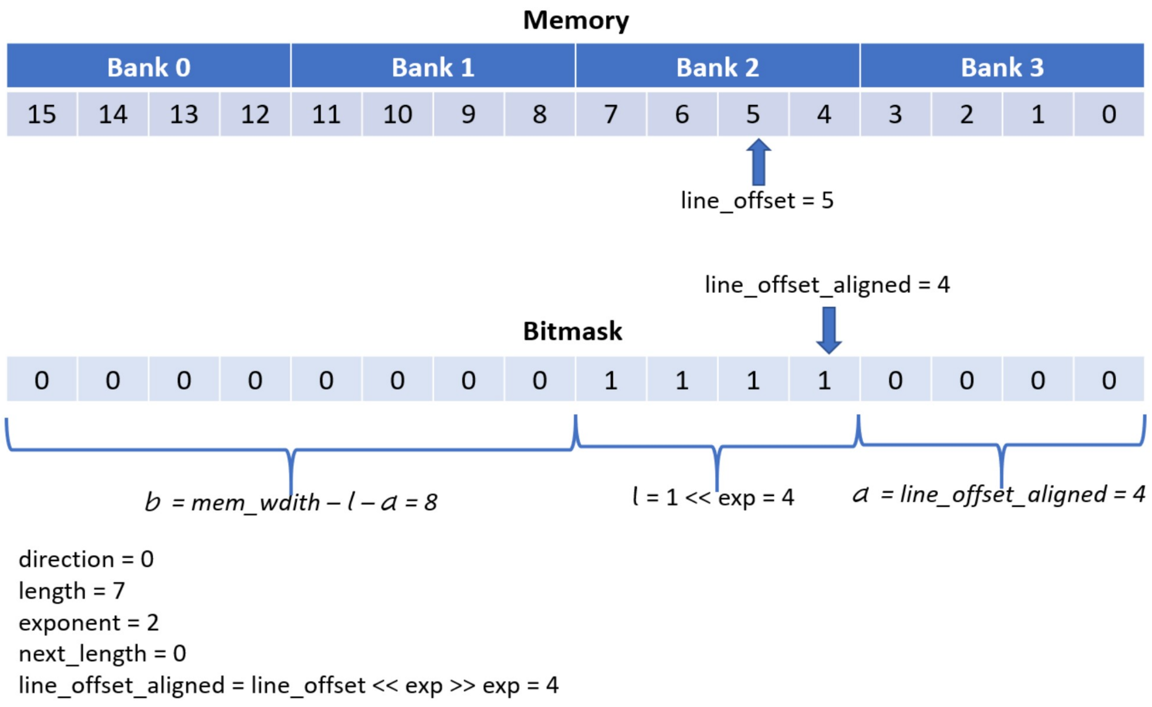

Figure 6: Example 3 shows the a, b, l computation for direction = constant

4 METHODOLOGY

We implemented both version of linear address bitmask generator and its testbench using Chisel3. The code was compiled, and Verilog code is generated. Then we used Synopsys Design Compiler [5] to complete the logical synthesis and frequency/power/area number are collected from synthesis report. The traditional local bitmask generation method is used as baseline. To be more specific, we compare the cell area, static power and frequency of linear address bitmask generators, since the goal is to achieve low power consumption and low area design with little frequency trade-off.

Our design consists of dynamic shifters and adders, both of them would increase in area and power with respect to the number of input bits. Since increasing memory width would increase the input bits of shifters and adders, we are also exploring the design space by changing the mem width parameter for both baseline and our design. To be more specific, we choose memory width equal to 16, 32, 64, 128 as testing cases. For each memory width, we are keeping the memory byte addressable, number of exponents as 4, memory size as 256 bytes. In addition to mem width, the number of exponents is also a determining factor, as the traditional design will increase linearly as the number of exponents increase. We choose the number of exponents to be 2, 3, 4, 5, 6 as a testing case, and compare our result to the baseline design. For each number of exponents, we keep the memory byte-addressable, memory with as 128-bytes and memory size as 256 bytes.

5 EVALUATION

As expected, area, power, increased by increasing memory width, and frequency decreased by increasing memory width. All simulation results are shown in Figure 9, and the data is shown in Appendix A. Shifters performance in area and power is better than baseline. Moreover, as the memory width increases, the performance in area and power gets better and better. For example, the shifter's area is 23.6% and 27.4% less than baseline design when the memory width is 16 and 128. And the shifter's power is 7.9% and 23% less than baseline design when the memory width is 16 and 128. Also, the shifter's frequency is 25% lower than baseline when memory width is 16.

However, when memory width increases to 128, the shifter's frequency is higher 5.7% than baseline design. Due to time constraints, we did not continue to increase the memory width of the experiment in this work. But we do not rule out that if we continue to increase the width, the shifter's frequency performance will be worse than baseline design. In general, in the experiment of increasing the width, our shifter's performance is better than baseline design. In the best case, compared with baseline design, the shifter's design can lower area, power and frequency performance by 27.4%, 23% and 25% respectively.

Figure 9 presents the simulation results for increasing the number of memory unit width exponent (number of exponent) as well. In the best case of area versus number of exponents, the shifter's area is 27.4% better than baseline design when the number of exponents is 4. In the best case of power versus number of exponents, the

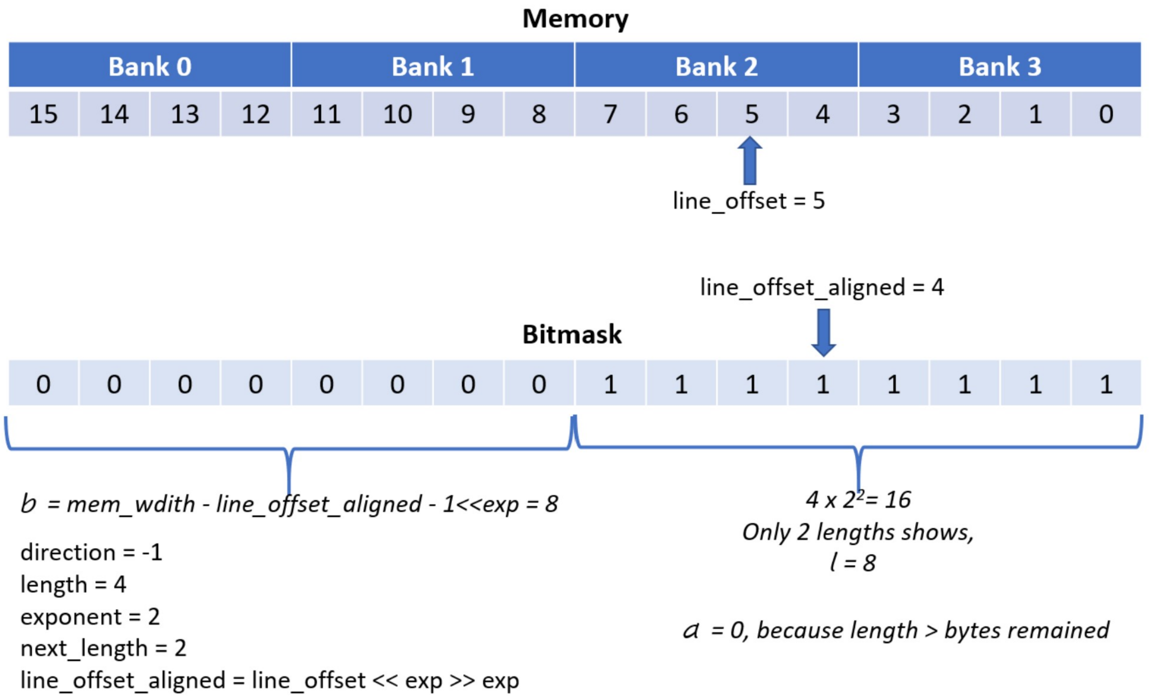

Figure 7: Example 4 shows the a, b, l computation for direction = descend

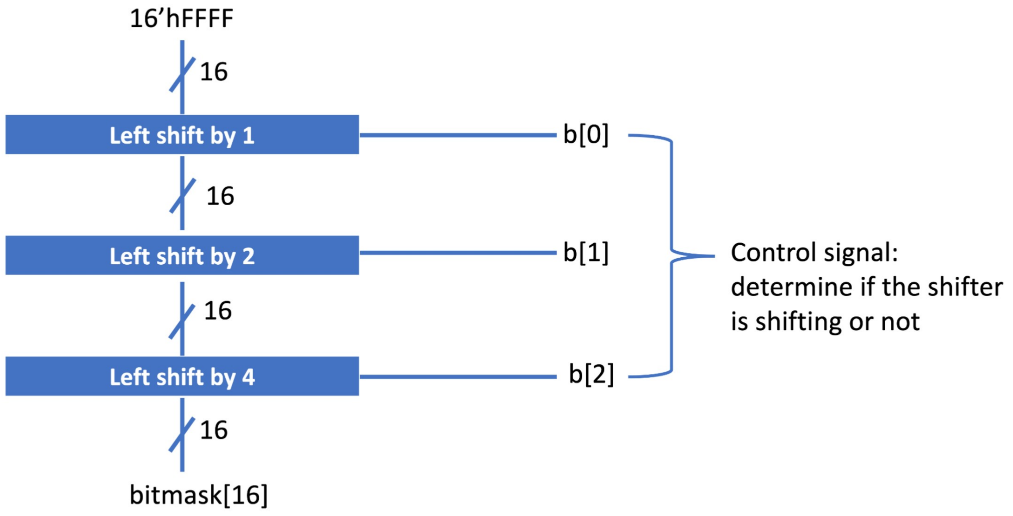

Figure 8: A Dynamic Shifter Example of Left Shifting 16'hFFFF by b Bit

shifter's power is 23.3% better than baseline design when the number of exponents is 5. In the best case of frequency versus number of exponents, the shifter's power is 28.4% better than baseline design when the number of exponents is 5. Overall, the shifter's design has better performance in area, power, with the best improvement of 27.4%, 23.3%.

6 CONCLUSION

Chisel makes the power of a modern software programming language available for hardware design, supporting parameterized generators and providing high-quality Verilog RTL output for ASIC flow synthesis. We have accomplished an alternative design compared to [7] for linear address bitmask hardware generators using Chisel. By using dynamic shifter and adders, our algorithm explores

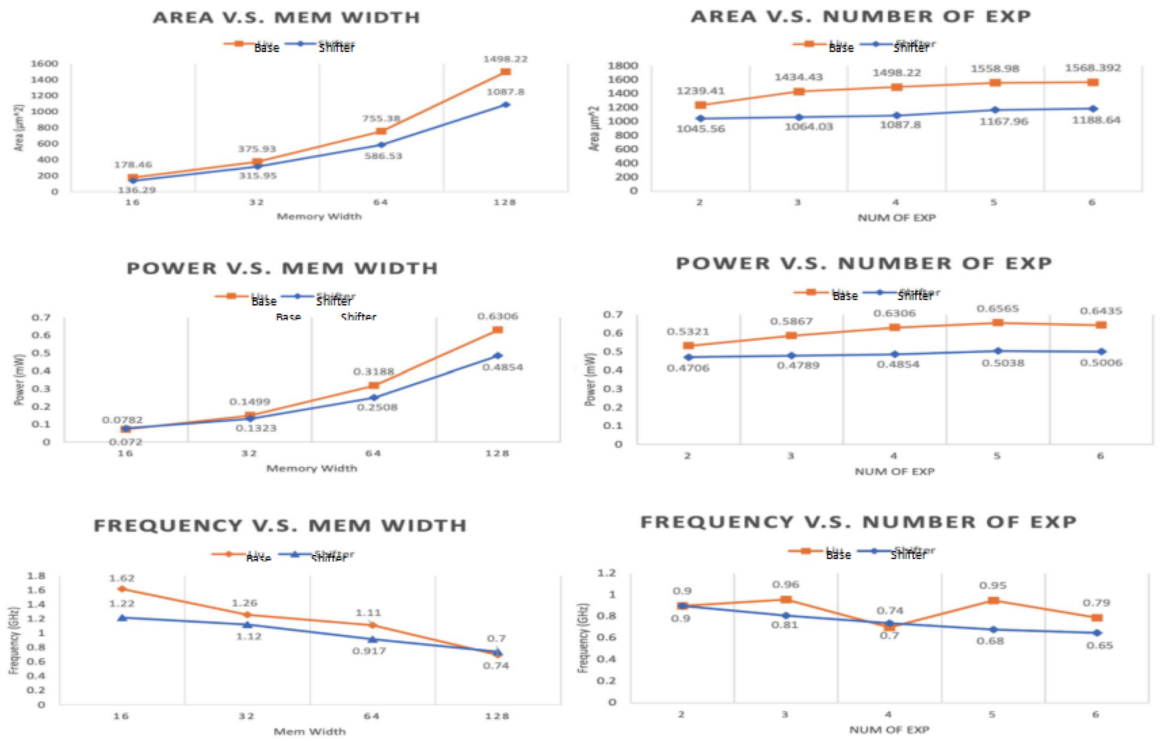

Figure 9: Comparison Between Baseline and Dynamic Shifter Design

possible trade-off between power, area, and frequency. After evaluation, our design is proven to be power efficient and area efficient with approximately 25% improvement than baseline design, while having little frequency loss under specific circumstances.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovi´c. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.
[2] Krochmalski, Jarosław. IntelliJ IDEA Essentials. Packt Publishing Ltd, 2014.
[3] Asanovic, Krste, *et al.* "The rocket chip generator." EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17 4 (2016).
[4] Gomez-Luna, Juan, *et al.* "Performance modeling of atomic additions on GPU scratchpad memory." IEEE Transactions on Parallel and Distributed Systems 24.11 (2012): 2273-2282.
[5] Kurup, Pran, and Taher Abbasi. Logic synthesis using Synopsys®. Springer Science & Business Media, 2012.
[6] Ercegovac, Milos D., and Tomas Lang. Digital arithmetic. Elsevier, 2004.
[7] Ramesh, Kini M., and David S. Sumam. "Comprehensive address generator for digital signal processing." 2009 International Conference on Industrial and Information Systems (ICIIS). IEEE, 2009.