

**Evaluation of the Effect of Saturation Arithmetic
and Other Techniques on Sum of Absolute
Difference (SAD) Computation in H.264**

Venkata Suman Sanikommu
sanikommu@wisc.edu

1. Introduction

Sum of Absolute Difference (SAD) technique is a very commonly used technique for motion estimation in various video encoding standards like H.264 and MPEG coding [1]. Motion estimation is a temporal prediction technique used in video encoding. The basic need for motion estimation stems from the fact that the consecutive video frames have similarities that in most cases can be intelligently exploited to reduce the number of bits in the file encoding process. Most of the consecutive video frames will be similar except for the changes that might be induced by objects moving within frames. The best case for motion estimation will be when the consecutive frames have no frame difference except for the differences caused by noise. If this is the case, it will be very easy for the encoder to efficiently predict the frame as a duplicate of the reference frame. Once encoder finds that the frames are same, the only information that needs to be transmitted to the decoder will be the syntactic overhead necessary to reconstruct the picture from the original reference frame. This sounds simple since we are saying that the two consecutive frames are exactly similar. But, in most of the cases, the frames differ at least by a few frames because of the objects moving across the frames in the video. This is when Sum of Absolute Difference calculation comes into play. It has been shown that this is one of the most effective techniques in motion estimation. The next few sections discuss about motion estimation and SAD in more detail.

The commonly used arithmetic technique in many coding standards is the use of modular arithmetic (sometimes called modulo arithmetic). In modular arithmetic, the numbers wrap around after they reach a certain value. Suppose if we are using 8 bits for the representation of a number in its binary form, if we try to add 1 to binary number 11111111, the actual value in decimal should have been 256. But in modular arithmetic, because 256 cannot be represented using 8 bits, it wraps around and this number will be represented as 0, the next number in a circular fashion after 255. Another technique that is gaining wide popularity now-a-days is the use of saturation arithmetic. Let us consider the same example that we discussed for modular arithmetic. Now, after addition, the value becomes 256 and hence can't be represented using 8 bits. Instead of wrapping

around, the value “saturates” at the highest value possible to represent using 8 bits, which in this example will be 255. So, even if you add any number to any number, the maximum possible number that can be represented using 8 bits in saturation arithmetic will be 255.

2. Background

2.1 MPEG Encoder

To understand the concept of video encoding and motion estimations, let's look at the MPEG coding. In MPEG [2, 3], video-sequences are compressed by exploiting both spatial and temporal redundancies. Spatial redundancies can be seen as small differences between local pixels and temporal redundancies can be seen as small differences between two temporally close video frames. These kinds of redundancies as mentioned in the earlier section can be exploited using predictive coding, but more compensation can be reached by using it together with motion estimation [4]. This is explained using the MPEG encoding process that is shown in figure 1.

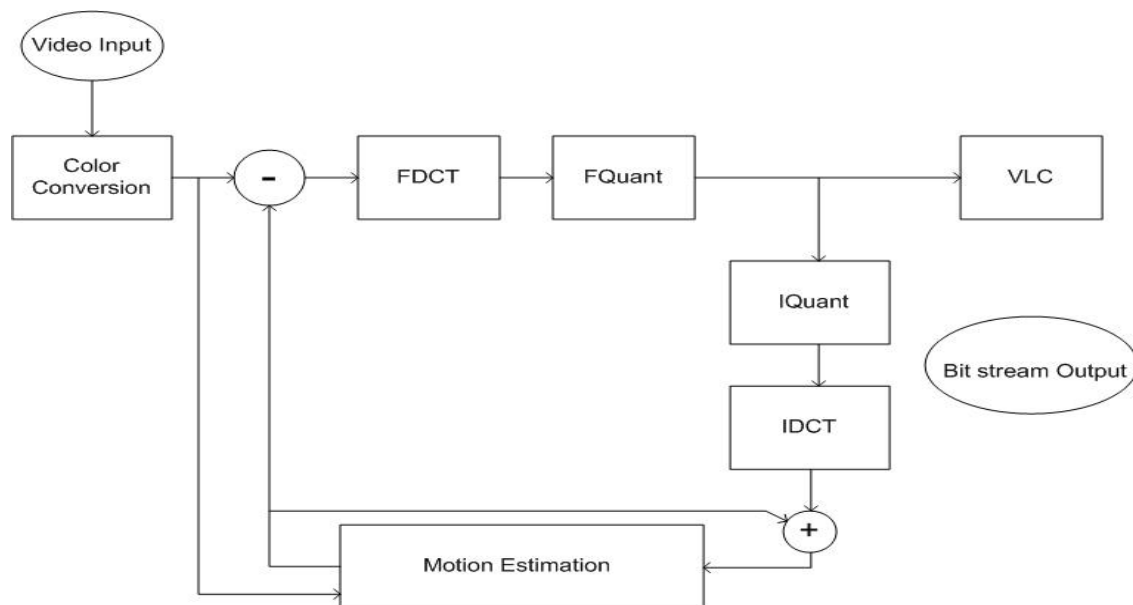


Figure 1: MPEG Encoder Implementation

In the diagram, FDCT denotes Forward Discrete Cosine Transformation, FQuant denotes the Forward Quantization and VLC denotes Variable Length Coding. The IQuant and IDCT are the inverse operations needed to reproduce the picture as it is available at the decoder. The motion estimation block uses these decoded images as reference instead of original images as the decoder has access only to these decoded images. In MPEG coding, there are two kinds of blocks: the 16x16 macro block and the 8x8 basic block. The basic block is used for DCT and the macro-block is used for motion estimation. Each of the steps depicted in figure 1 are explained here.

Color Conversion: The input color-space is transformed into the YCbCr color-space. The chrominance are sub-sampled by a factor of two in both horizontal and vertical direction. Thus a 16x16 block from the video signal results in four 8x8 luminance blocks, one 8x8 C_b block and one 8x8 C_r block. These 8x8 blocks are used by DCT and the 16x16 luminance block will be used by motion estimation.

Motion Estimation: For each 16x16 block luminance pixels in the current frame, a motion vector is computed. This motion vector contains the relative position of the block most closely resembling the current block in the reference frame.

DCT: DCT is performed on each 8x8 blocks which can be either blocks from the frame or only the difference blocks.

Quantization: DCT coefficients computed in the DCT process are quantized. If the quantization is too coarse, the resulting video quality might be lower.

Variable Length Coding: The results of the quantization process are serialized into a bit stream using run-length coding and variable length coding.

2.2 Motion-Vector Search

Several algorithms are available to compute which block in the reference frame most closely resembles the current block. One technique is the exhaustive search which is time consuming but will provide us with the best of the possible results. Others are using

heuristic-based algorithms which are much faster at the cost of a possibly less optimal result.

Irrelevant of the search algorithms used, a metric is used which indicates the closeness between compared blocks. The two most common metrics found in different search algorithms are the mean square error (MSE) and the mean absolute difference (MAD). MSE can be performed by the following equation:

$$MSE(x, y, r, s) = \frac{1}{256} \sum_{i=0}^{15} \sum_{j=0}^{15} (A_{(x+i, y+j)} - B_{((x+r)+i, (y+s)+j)})^2$$

The MAD is performed by

$$MAD(x, y, r, s) = \frac{1}{256} \sum_{i=0}^{15} \sum_{j=0}^{15} |A_{(x+i, y+j)} - B_{((x+r)+i, (y+s)+j)}|$$

MAD can also be written as

$$MAD(x, y, r, s) = \frac{SAD(x, y, r, s)}{256}$$

Where SAD is the summation of absolute differences, given by

$$SAD(x, y, r, s) : \sum_{i=0}^{15} \sum_{j=0}^{15} |A_{(x+i, y+j)} - B_{((x+r)+i, (y+s)+j)}|$$

In these equations (x, y) is the position of the current block and (r, s) denotes the motion vector, i.e. the displacement of the current block (A) relative to the block in the reference frame (B).

2.3 Computing the Sum Absolute Difference (SAD)

A direct approach to compute the SAD value consists of the following steps:

- Compute $(A_i - B_i)$ for all 16x16 pixels in the two blocks A and B

- Determines which $A_i - B_i$ is less than zero and produce in that case $B_i - A_i$ as the absolute value, else compute $A_i - B_i$.
- Perform the accumulate operation to all 16x16 absolute values.

These are the three steps in calculating the SAD values. Many techniques to speed up the computation of SAD are proposed [5], but here are not discussed due to the scope of the project.

2.4 Saturation Arithmetic

Saturation arithmetic started to gain popularity in the recent days because the use of saturation arithmetic can reduce the computation complexity. But, this reduction in computation complexity is achieved only at the expense of the resultant values being “approximates”. Sometimes the saturated values might result in the cause of errors because of the inaccuracy in the results. As we have seen in one of the questions in the final exam where we were asked to comment on which one would one prefer (IIR or FIR) to use saturation arithmetic, such decisions need to be taken before we apply saturation arithmetic to the circuit or algorithm of our interest. Many assembly level languages like MMX started to support saturation arithmetic[8, 9].

3. Saturation Arithmetic – SAD

The idea used in this paper was to use saturation arithmetic and see if we can reduce the complexity of the SAD computation and make sure that we are not losing the accuracy in finding the best matching block from the reference frame. Just to explain the concept using an example, consider a frame with 100 blocks that can be searched to find a match from a reference frame. If we don't use saturation arithmetic, the maximum value of SAD can reach up to a value that will use 16 bits. But, in most of the cases, the SAD value won't be as high a value to use 16 bits. If there is no difference between two successive frames, the value will be 0 (ideally). So, we would need anywhere between 0 to 16 bits to represent the SAD values. Suppose if we have a set of SAD values {6367, 9573, 2184,

12353}, the one with the best matching block will be one with the minimum SAD value and hence in this case it will be block that gave a SAD value of 2184. Now, if we use saturation arithmetic with 13 bits to represent, we can see that some of these SAD values will saturate resulting in a new set of SAD values: {4095, 4095, 2184, 4095}. From this example it can be seen that since the minimum SAD value is less than the maximum possible SAD value with saturation, we still found the best block even though the other SAD values saturated. Now, consider the case if we use only 12 bits with saturation. The new SAD values will be {2047, 2047, 2047, 2047}. Now, this is a difficult case as it can be seen that all the SAD values saturated and hence are same. Now, finding the best matching block will be difficult. The encoder will pick one of these four blocks randomly and uses it for compression. If the selected block is not a best match, we lose a significant portion of video quality. The chance that the selected block is the perfect match is low in this case. So, this is the problem that was addressed here in this paper.

4. Experiments and Results

The H.264/AVC reference software used for this project was the JM10.2 version. Here when discussing the results, there are two important things that need to be noted, otherwise they might be confusing. Where ever, the term “SAD value” is mentioned, it is the already found minimum SAD value from each of the individual block matching and is used in comparing the results of SAD from an entire encoding process instead of just a particular block. Another term is the “Individual SAD”, which essentially means the SAD of an individual block and not considering all the SAD values in the whole picture. Noting these two differences will help you understand the results better. In experimenting with saturation arithmetic for SAD, the three input files available from the reference software, namely `foreman_part_QCIF.yuv`, `foreman_part_QCIF_422.yuv` and `foreman_part_QCIF_444.yuv` are used. Effort was made to get some more input files to get more ‘finer’ results, but was unable to find more input files.

Figure 2 shows the distribution of SAD values (note that these are from the whole encoding process and not for individual block matching). Essentially what this figure

explains is that the percentage of SAD values out of all the SAD values found that can be represented using a particular number of bits. For example, let us consider the foreman_part_QCIF.yuv from figure 2. It shows that for this particular input, 100% of all the minimum SAD values can be represented using 15 bits itself. As we reduce the number of bits that we use to represent, the number falls as shown. (Only 80% can be represented using 14 bits and only 37% can be represented using 13 bits). Thus this figure shows us what the range of the minimum SAD values is. So, for the three inputs considered, the behavior of SAD values changes only by a small amount, all in the range of around 200 to about 32000. To be exact, here are the minimum and maximum of the ‘minimum SAD’ values found from these inputs.

Input File	Minimum of min{SAD}	Maximum of min{SAD}	Minimum number of bits required
Foreman_part_QCIF.yuv	397	31841	15
Foreman_part_QCIF_422.yuv	357	30941	15
Foreman_part_QCIF_444.yuv	176	31841	15

Table 1: Minimum and Maximum of ‘minimum or best’ SAD values

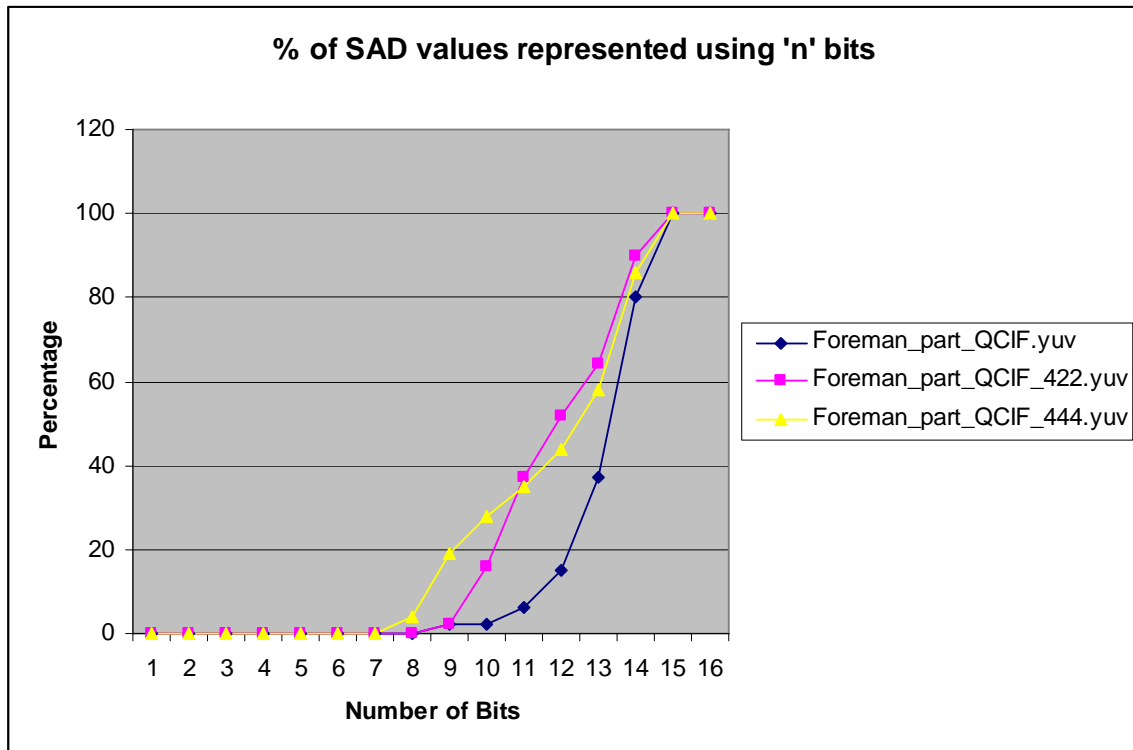


Figure 2: Percentage minimum SAD values that can be represented with ‘n’ bits without saturation

If saturation is used, the results were as expected and the graph looks as if it was a reverse of figure 2. (Reverse in the sense that the percentage of values saturated will be $(100 - \text{percentage of values that can be represented with } n \text{ bits})$). The results when saturation is used are shown in figure 3.

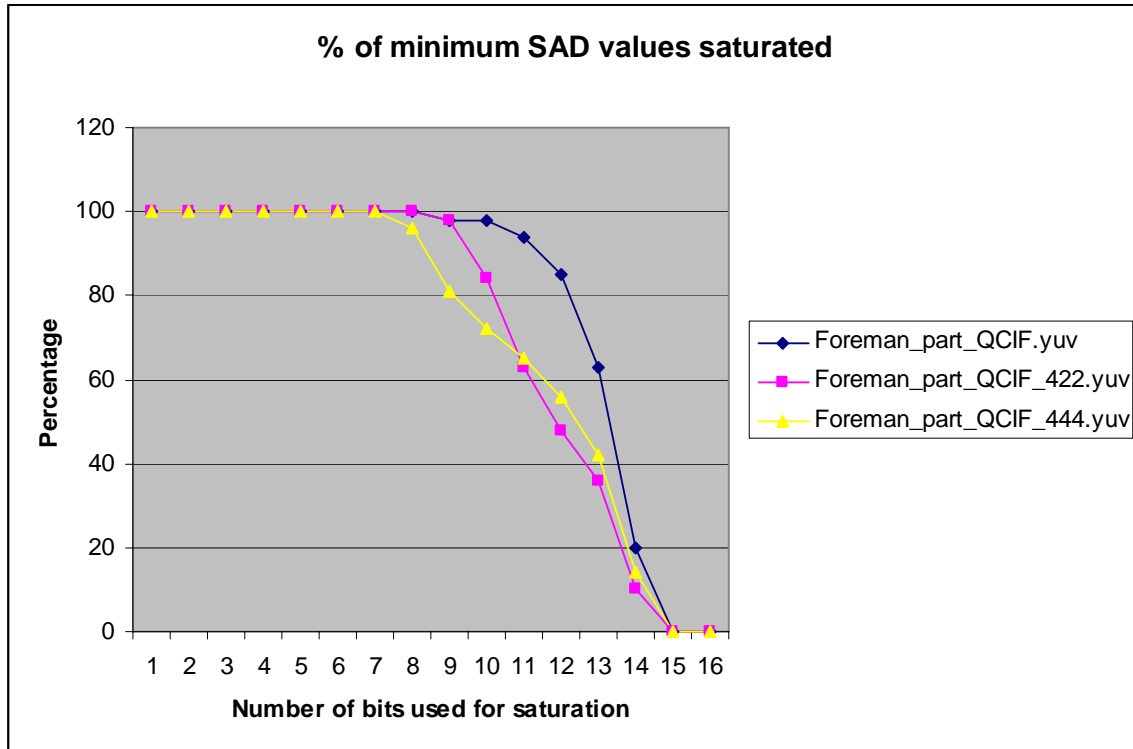


Figure 3: Percentage minimum SAD values that will saturate when ‘n’ bits are used

It can also be understood that if the minimum of the SAD values saturates, then the rest of the individual SAD values will also saturate. (If minimum is the saturated values, then the rest should also have saturated, otherwise it won’t make sense). This property was also observed in the output files from the H.264 encoder.

As was actually proposed, I also wanted to see how the video quality will be affected by the use of saturation arithmetic. But in order to do such kind of a quality analysis, a very complex programming model needs to be developed which will read in the entire video and then analyze the error in it because of the improper SAD values as an affect of saturation. Also, I wanted to see saturation’s affect on encoded file size. But it was observed that the encoded file size doesn’t change but the quality of the video can only be affected. The reason for this was that no matter whether the matched block is the best one

or a random one, its size in terms of bytes remains constant because of the standard size of each block. So, I am unable to provide any analysis in that respect.

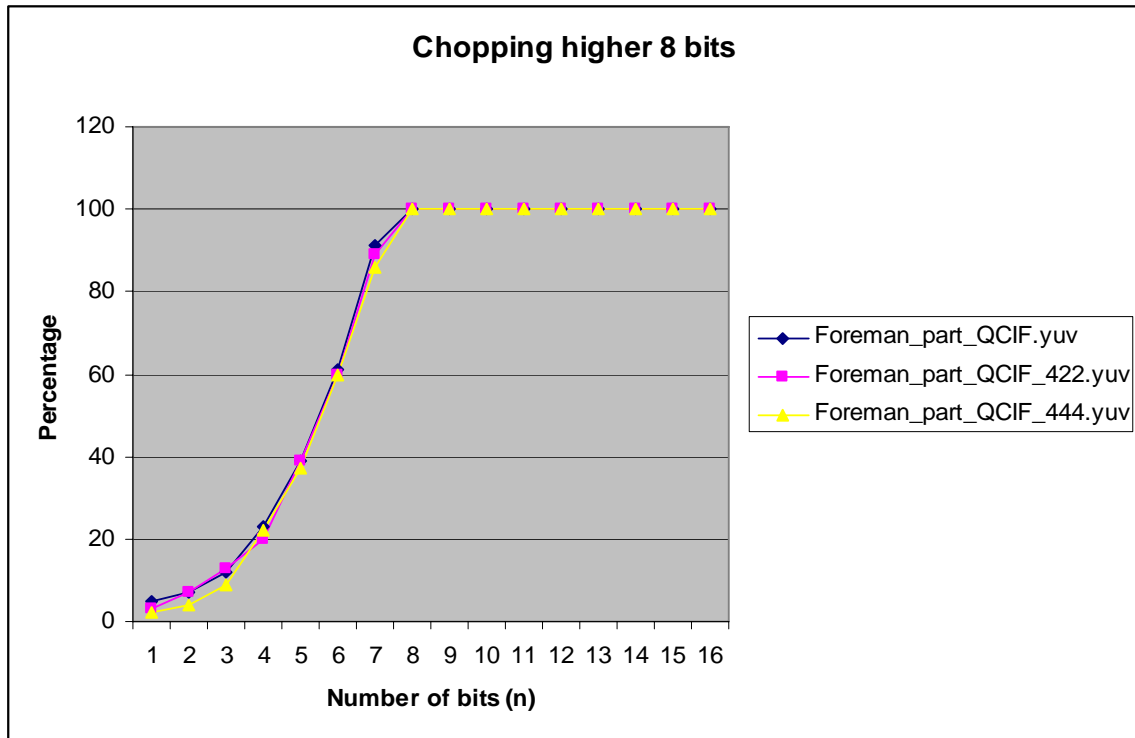


Figure 4: Distribution of SAD value if higher 8 bits are chopped off

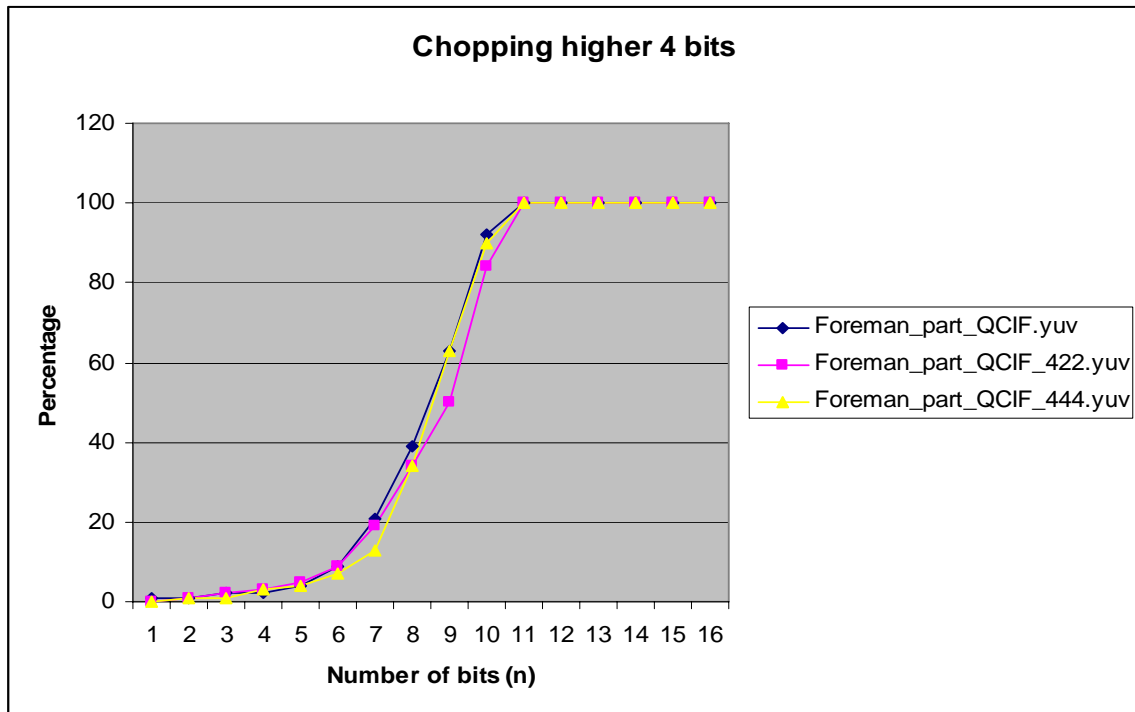


Figure 5: Distribution of SAD values if higher 4 bits are chopped off.

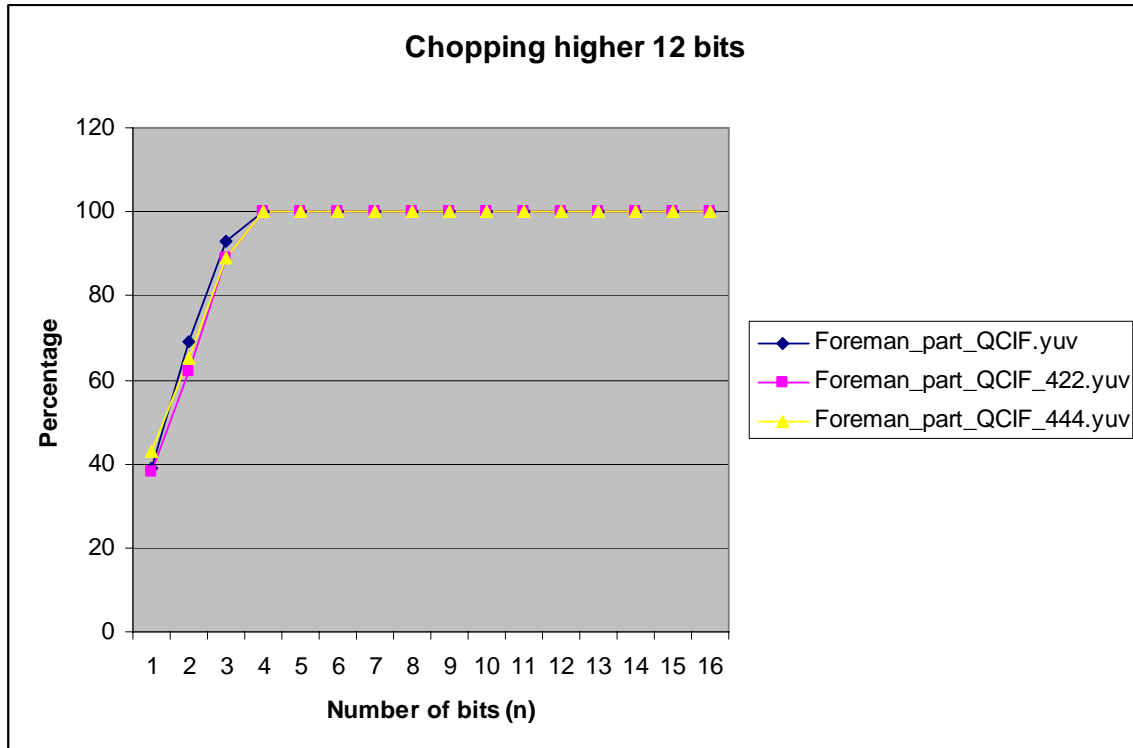


Figure 6: Distribution of SAD values if higher 12 bits are chopped off.

Other ideas advised by professor Hu to chop off the lower or higher k bits are also tried. Figures 4, 5 and 6 show the effect of chopping off higher k bits on SAD values. But the affect of these techniques on the encoding was not studied because of the unavailability of software that will help me do that. Theoretically, if we chop off the higher k bits, the affect will be random as the lower $n-k$ doesn't necessarily indicate whether that value was the minimum or not. So, this again comes down to a random selection of matching blocks as the one with lesser lower $n-k$ bit value doesn't necessarily be a best matching block. As it can be seen from the figures, the behavior will be random and the curves are justing shifting left as we chop more and more number of bits. Figure 7, 8 and 9 show the effect of chopping off lower k bits. Here again, if we chop off lower k bits, the minimum will still be a minimum unless the SAD value is less than 2^k in which case we will have the same affect as if we were saturating. Moreover, chopping lower few bits will not reduce complexity as this can be done only after the SAD values are computed and actually add latency for calculating the value after division.

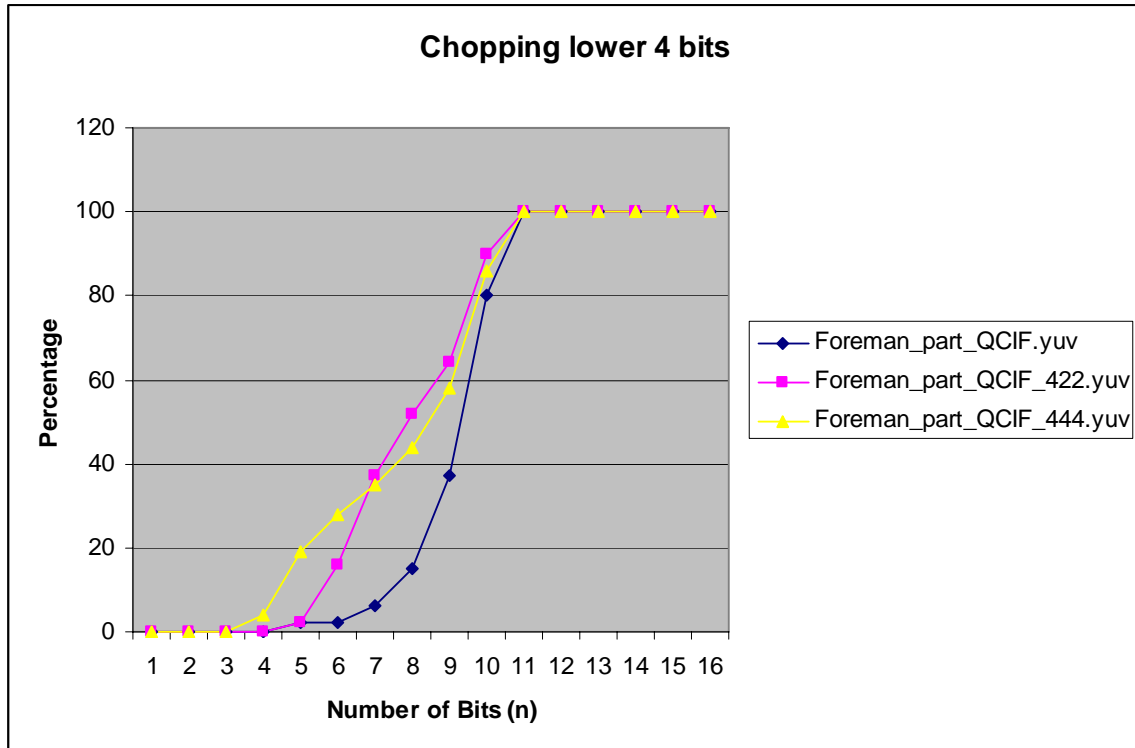


Figure 7: Distribution of SAD values if lower 4 bits are chopped off.

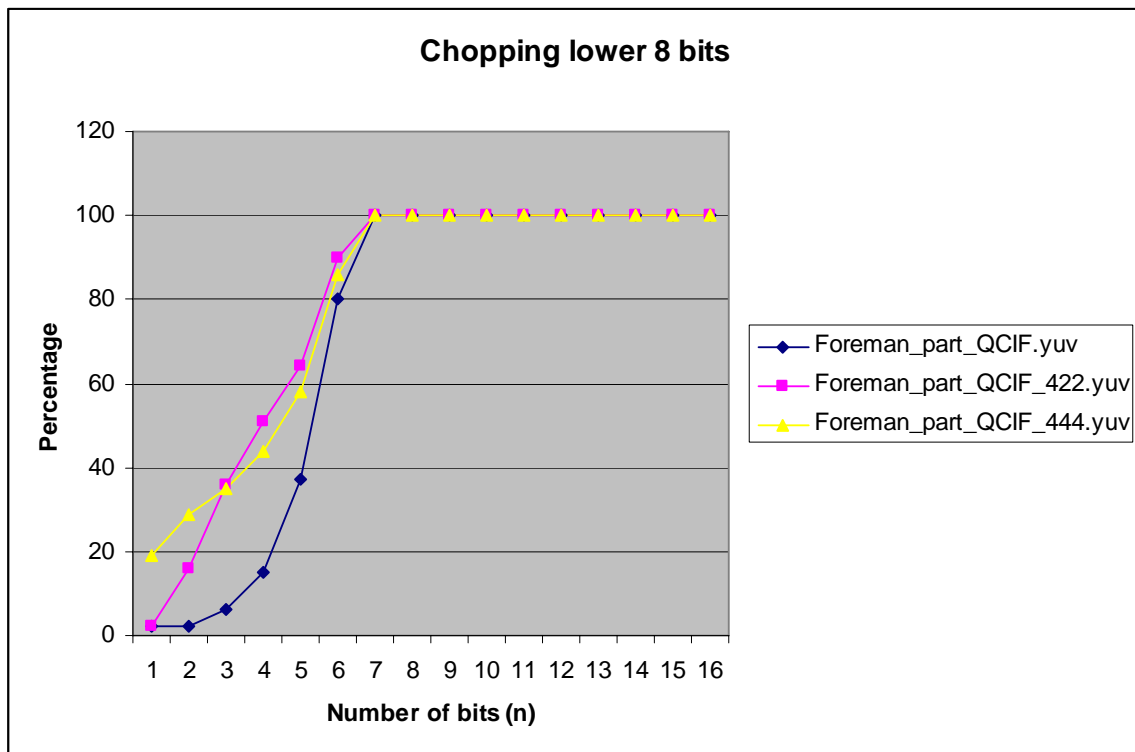


Figure 8: Distribution of SAD values if lower 8 bits are chopped off.

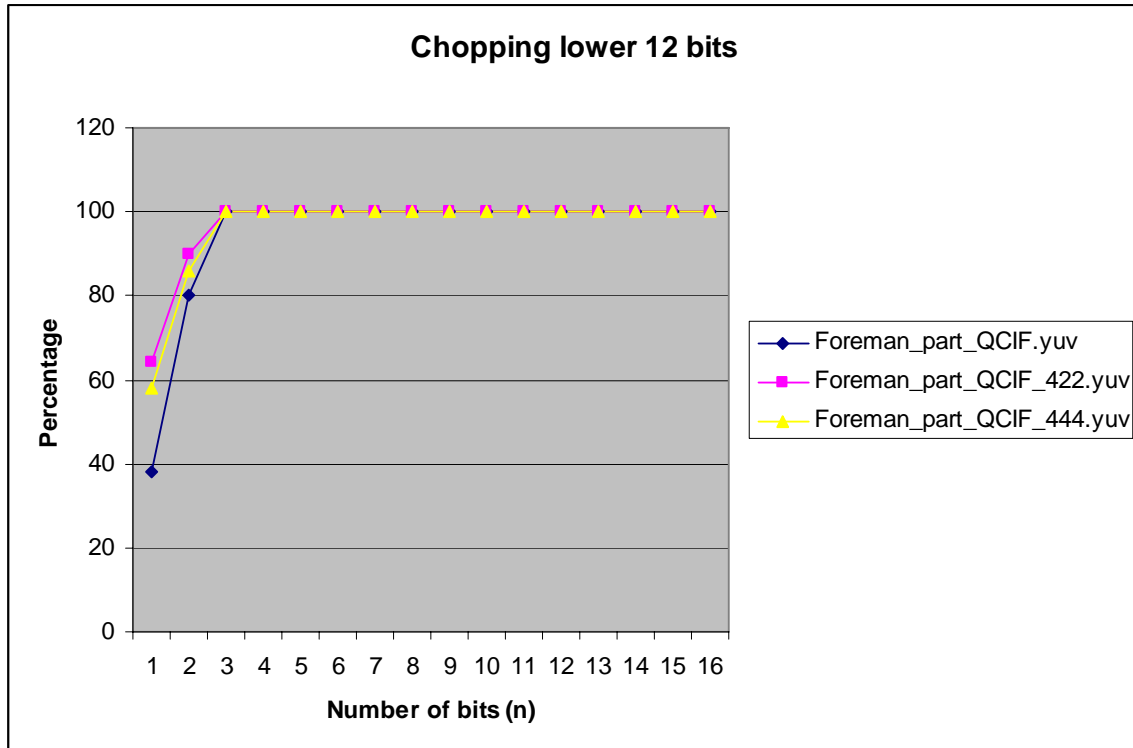


Figure 9: Distribution of SAD values if lower 12 bits are chopped off

5. Conclusion

From the experiments that I have done here, it can be concluded that the use of saturation arithmetic or any other bit manipulation techniques will result in a random and unpredictable SAD calculations. Saturation arithmetic should not be used unless one knows that the SAD values cannot be more than a particular number (say, if the video stream is assumed to change very infrequently). If saturation arithmetic is used without proper knowledge of the video streams being encoded, it will result in a very poor video quality. So, use of saturation arithmetic is a decision that needs to be taken depending on the streams for which it is going to be used. Also, I would have been able to present more realistic results if there was a way to analyze the encoded bit stream (Other than visual checks). Future work on this project would be to write or get hold of a simulator using which we can analyze the actual effects of these techniques from the encoded bit stream.

6. References

- [1] T. Wiegand, G. J. Sullivan, G. Bjntegaard and A. Luthra, “*Overview of the H.264/AVC Video Coding Standard*”, IEEE Transaction on Circuits and Systems for Video Technology, July 2003.
- [2] J. L. Mitchell, W. B. Pennebaker, C. E. Fogg, and D. J. LeGall, “*MPEG Video Compression Standard*”, Digital Multimedia Standard Series, Chapman and Hall, 1996.
- [3] D. L. Gall, “*MPEG: A Video Compression Standard for Multimedia Applications*”, Communications of the ACM, 34(4):46-58, April 1991.
- [4] S. Kappagantula and K. Rao, “*Motion Compensated Predictive Coding*”, Proceedings of International Symposium. SPIE, San Diego, CA, August 1983.
- [5] S. Vassiliadis, E. A. Hakkened, J. S. S. M. Wong and G. G. Pechanek, “*The Sum-Absolute-Difference Motion Estimator Accelerator*”, Proceedings of the 24th Euromicro Conference, 2000.
- [6] G. A. Constantinides, P. Y. K. Cheung and W. Luk, “*Synthesis of Saturation Arithmetic Architectures*”, ACM Transactions on Design Automation of Electronic Systems (TODAES), 2003.
- [7] N. Yadav, M. Schulte and J. Glossner, “*Parallel Saturating Fractional Arithmetic Units*”, Proceedings of Ninth Great Lakes Symposium on VLSI, 1999.
- [8] R. B. Lee, A. M. Fiskiran, “*PLX: A Fully Subword-Parallel Instruction Set Architecture for Fast Scalable Multimedia Processing*”, Multimedia and Expo, 2002. ICME’02.
- [9] A. Peleq, U. Weiser, “*MMX Technology Extension to the Intel Architecture*”, Micro, IEEE, August 1996.

7. Appendix

7.1 Code snippet that was modified in file macroblock.c of encoder in H.264

```
for (j=0;j<4;j++)
    for (i=0;i<4;i++)
        M4[j][i]=M0[0][i][0][j]/4;
for (j=0;j<4;j++){
    M3[0]=M4[j][0]+M4[j][3];
    M3[1]=M4[j][1]+M4[j][2];
    M3[2]=M4[j][1]-M4[j][2];
    M3[3]=M4[j][0]-M4[j][3];
    M4[j][0]=M3[0]+M3[1];
    M4[j][2]=M3[0]-M3[1];
    M4[j][1]=M3[2]+M3[3];
    M4[j][3]=M3[3]-M3[2];
}
for (i=0;i<4;i++){
    M3[0]=M4[0][i]+M4[3][i];
    M3[1]=M4[1][i]+M4[2][i];
    M3[2]=M4[1][i]-M4[2][i];
    M3[3]=M4[0][i]-M4[3][i];
    M4[0][i]=M3[0]+M3[1];
    M4[2][i]=M3[0]-M3[1];
    M4[1][i]=M3[2]+M3[3];
    M4[3][i]=M3[3]-M3[2];
    for (j=0;j<4;j++)
        current_intra_sad_2 += absm(M4[j][i]); // Absolute values are being accumulated
}
if(current_intra_sad_2 > MAX_VALUE){ // Saturating the value depending on n
bits
```



```

my $k;
my $prev_min;
my $prev_max;
$total_sad_values = 0;
$prev_min = 32768;
$prev_max = 0;
while($current_line = <IN>){
    if($current_line =~ /BEST/){
        @temp = split(" ", $current_line);
        $total_sad_values++;
        for($i=1;$i<17;$i++){
            if($temp[2] < (2**$i)){ # < gives us one kind of values and >= gives another
kind of values
                $stats[$i]++;
            }
        }
        if($temp[2]<$prev_min){
            $prev_min = $temp[2];
        }
        if($temp[2]>$prev_max){
            $prev_max = $temp[2];
        }
    }
}
for($i=1; $i<17;$i++){
    $j = sprintf("%.0f", $stats[$i]*100/$total_sad_values);
    $k = 2**$i;
    print "\nPercent SAD values less than $k\t: $j"; }
print "\nHighest SAD Value\t: $prev_max";
print "\nLowest SAD Value\t: $prev_min";
print "\nTotal SAD values found\t: $total_sad_values";

```

Webpage

<https://mywebspace.wisc.edu/sanikommu/web/academic/ece734/734HW.htm>