Implementation of H.264/MPEG-4 AVC Decoder

Fanyu Ran[a], Yang Zhou[b], and Yifei Zhou[c]

[a]Student No.8759122, E-mail: fran027@uottawa.ca, University of Ottawa
[b]Student No.8657223, E-mail: yzhou152@uottawa.ca, University of Ottawa
[c]Student No.8635051, E-mail: yzhou151@uottawa.ca, University of Ottawa

ABSTRACT

The target of this project is to implement a minimal H.264 Baseline/Constrained Baseline Profile decoder in Python programming language. The product is expected to be capable to perform the following tasks:

- BitStream parsing
- Inverse quantization
- Inverse transformation
- Motion compensation

and, as a result, decode a displayable video sequence from a compressed video file.

Keywords: H.264 decoder Python

1. INTRODUCTION

H.264/AVC is the latest in a series of standards published by the ITU (International Telecommunications Union) and ISO (International Standards Organization). It describes and defines a method of coding video that can give better performance.

H.264 makes it possible to compress video into a smaller space, which means that a compressed video clip takes up less transmission bandwidth and/or less storage space compared to older codecs. It can be applied to transmit HD television over a limited-capacity broadcast channel, to record hours of video on a flash memory card and to deliver massive numbers of video streams over an already busy Internet.

2. PYTHON PROGRAMMING LANGUAGE OVERVIEW

We chose Python as the main programming language to implement this project.

Python is a widely used high-level, general-purpose, interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than possible in languages such as C++ or Java. [*]

Although programs written in C/C++ mostly have significantly better performance then ones in Python, they have to pay for verbosity as trade-off. Due to the statement from Python's official website, Python code is typically 5-10 times shorter than equivalent C++ code. [†]

The dynamic property and modern syntax of Python can eliminate much more noise in the code, making the idea behind the program more clear. For example, to calculate Fibonacci sequence in plain iterative method, we can implement it as below:

C:

[*] From Wikipedia: https://en.wikipedia.org/wiki/Python_(programming_language)

[†] From Python official website: https://www.python.org/doc/essays/comparisons

```
long long int fibb(int n) {
    int fnow = 0, fnext = 1, tempf;
    while(--n>0){
        tempf = fnow + fnext;
        fnow = fnext;
        fnext = tempf;
        }
        return fnext;
}
```

Python:

```
def fibIter(n):
    if n < 2:
        return n
    fibPrev = 1
    fib = 1
    for num in xrange(2, n):
        fibPrev, fib = fib, fib + fibPrev
    return fib
```

As a project which aims to be a concept prototype, we care more about readability and productivity instead of performance. Python works well for this target.

3. H.264 DECODING OVERVIEW

3.1 H.264 Profiles

H.264 specification includes several profiles, each specifying a subset of the optional features in the H.264 standard. A Profile places limits on the algorithmic capabilities required of an H.264 decoder. Each Profile is intended to be useful to a class of applications. For example, the Baseline Profile may be useful for low-delay, such as video conferencing, with relatively low computational requirements. The Main Profile may be suitable for basic television/entertainment applications such as Standard Definition TV services. The High Profiles add tools to the Main Profile which can improve compression efficiency especially for higher spatial resolution services, such as High Definition TV.

3.2 Main Decoding Processes

The decoding process is carried out by an H.264 video decoder. Flow diagram is shown in Figure 1.

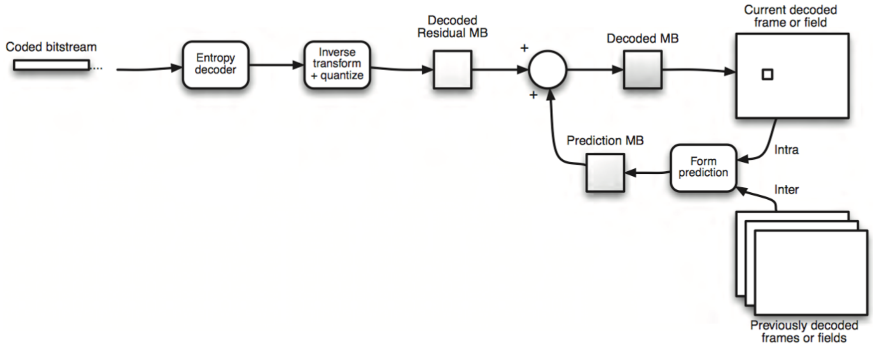

Figure 1. H.264 decoding process[1]

Initially, decoder codes compressed bitstream and does entropy decoding, obtaining a coefficient matrix. The following steps are dequantization and IDCT. Residual is output of this stage. Then the decoder will decode the frame depending on the combination of residual and prediction results. Notice that, in the both of coder and decoder, data is processed in units of macroblock. A decoded residual macroblock is formed by re-scaling and inverse transformation. Prediction is created in decoder and is added to the residual to generated a decoded macroblock.

3.2.1 Parsing Process with Entropy Decoding

As shown in Figure 2, inputs to parsing process are bitstream from contents of video slices. Output of this process are syntax element values. (Video sequence that is represented in a particular format which is syntax).

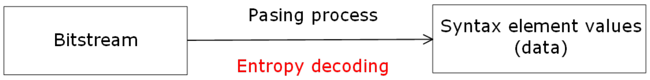

Figure 2. Entropy decoding process

Actually, several methods are specified by H.264 for coding symbols, including Fixed Length Code, Exponential-Golomb variable length code, CAVLC and CABAC.

And the data can be divided into two parts: decoder controlling parameters and transform coefficients. Above the slice data level, bits are decoded using Fixed Length decoder or Exp-Golomb decoder. In the slice data level and below, coefficient values are decoded in one of two ways. In CABAC mode, all symbols are decoded by CABAC decoder. Otherwise, coefficient values using CAVLC and other symbols are coded using fixed length or Exp-Golomb decoder. In our baseline decoder, Exp-Golomb and CAVLC are utilized.

3.2.2 Exponential Golomb Decoder

Exponential Golomb codes (ExpG) are an efficient way of representing data symbols with varying probabilities. It assigns short codewords to frequently-occurring data symbols and long codewords to less common data symbols. A parameter *code_num* indicates codewords. The length of Exponential-Golomb codes are varying with two properties:

1. With the increase of *code_num*, code length also rises.
2. Without looking up for tables, each code can be generated and decode algorithmically.

The structure of an Exp-Golomb codeword:

[Zero prefix] 1 [Information]

The codeword consists of M zero prefix ($M \geq 0$, M is an integer), a one and M-bit information. The length of the codeword is $2M + 1$ bit.

The process of decoding *code_num* is followed:

1. The decoder reads M consecutive zeros and calculates the total length of the next Exp-Golomb codeword as $2M + 1$ bits.
2. Read a one and ignore it.
3. It reads the remaining M bits and calculates *code_num* which is then mapped to k (data).

There is an example:

$$codeword = \underbrace{0000000}_{\text{Zero Prefix}}\ \mathbf{1}\ \underbrace{1100011}_{\text{Info.}}$$

The codeword to be decoded has 7 leading zeros and a one, followed by the information code 1100011 which will be converted to decimal number. Then we calculate the result *code_num* depending on the formula:

$$code\_num = 2M + \text{Information} - 1$$

Then *code_num* is mapped to k (data) according to different mapping types which is shown in Table 1.

Mapping type	Description
ue	Unsigned direct mapping. $$code\_num = k$$ Used for macroblock type, reference frame index and others.
te	Truncated mapping: if the largest possible value of k is 1, then a single bit b is sent where $b =!code\_num$, otherwise *ue* mapping is used.
se	Signed mapping, used for motion vector difference, delta QP and others. k is mapped to *code_num* as follows: $$codenum = 2\vert k\vert \quad (k \leq 0)$$ $$codenum = 2\vert k\vert - 1 \quad (k > 0)$$ *code_num* is mapped to k as follows: $$k = (1)^{code\_num+1} ceil(code\_num/2)$$
me	Mapped symbols, k is mapped to *code_num* according to a table specified in the standard.

Table 1. Code number mapping types[1]

3.2.3 Context Adaptive Variable Length Coding (CAVLC)

Residual, the transform coefficients are decoded by CAVLC decoder. Each 8×8 block of quantized transform coefficients is processed as four 4×4 blocks for the purposes of CAVLC encoding and decoding. The decoding process can be divided into 5 stages:

1. Decode the number of coefficients and trailing ones
2. Decode the sign of each trailing ones
3. Decode the levels of the remaining non-zero coefficients
4. Decode the number of zeros preceding each non-zero coefficient
5. Zero padding

Decode the number of coefficients and trailing ones, according to input bitstream and standard table 9-5. Then the result is used to decode sign of trailing ones. To decode the levels of the remaining coefficients, we need to obtain the suffix length and get level prefix depending on standard table 9-6. The next step is to decode the number of zeros preceding each non-zero coefficient. The total number of coefficients and standard table 9-7 are needed. After the four steps, use zero to pad the remaining elements.

3.3 Prediction Overview

The prediction methods may have a great influence on the compression performance.H.264 supports two prediction options: **Intra prediction** using data within the current frame, **Inter prediction** using motion compensated prediction from previously coded frames. H.264 provides multiple prediction block sizes, multiple reference frames and special modes. All these features give H.264 a great deal of flexibility in the prediction process. By selecting the best prediction options for an individual macroblock, H.264 can minimize the residual size to produce a highly compressed bitstream.

3.3.1 Macroblocks In Prediction

H.264 divides each frame in the video into macroblocks, which are blocks of 16x16 pixels. The macroblocks are decoded from left to right, and then top to bottom. There are three types of marcoblocks for prediction sources, an I Macroblock, a P Macroblock and a B Macroblock. An I Macroblock (I MB) is predicted using intra prediction from neighboring samples in the current frame. A P Macroblock (P MB) is predicted from samples in a previously-coded frame which may be before or after the current picture in display order. Each partition in a B Macroblock (B MB) is predicted from samples in one or two previously-coded frames. (Figure 3)

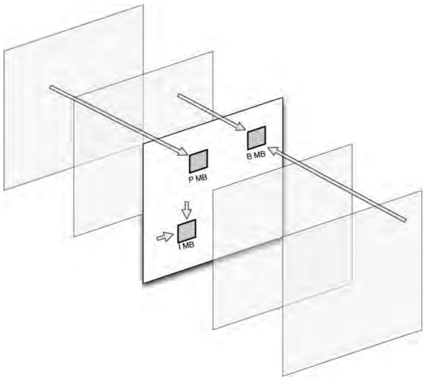

Figure 3. Marcoblock types[1]

3.3.2 Intra-Prediction

Intra prediction is used to use part of the frame to predict the other parts. An I Macroblock (I MB) is predicted using intra prediction from neighbouring samples in the current frame. For every block of the frame up to 16x16 pixels, intra-prediction uses previously decoded neighbor blocks to give an estimate for the new block.

In an intra macroblock, there are three choices of intra prediction block size for the luma component, namely 16×16, 8×8 or 4×4. A single prediction block is generated for each chroma component. Each prediction block is generated using one of a number of possible prediction modes.(Figure 3.3.2)

Intra prediction block size	Notes
16×16 (luma)	A single 16×16 prediction block P is generated.
8×8 (luma)	An 8×8 prediction block P is generated for each 8×8 luma block.
4×4 (luma)	A 4×4 prediction block P is generated for each 4×4 luma block.
Chroma	One prediction block P is generated for each chroma component. Four possible prediction modes. The same prediction mode is used for both chroma components.

Table 2. Intra prediction types[1]

4×4 luma has 9 prediction modes, which are shown in Figure 4. The samples, labeled A-M, have previously been decoded and reconstructed and they act as the prediction reference, then decoder copies the reference samples to the currently decoding blocks to continue the decode process. and decoder to form a prediction reference.

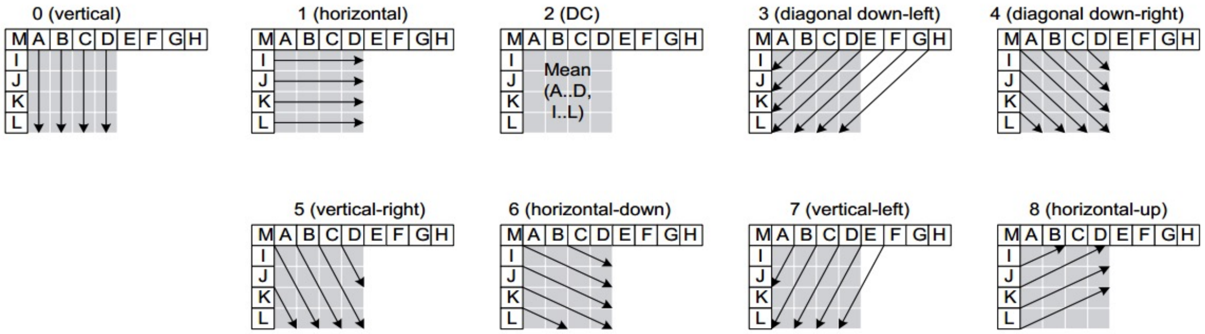

Figure 4. 4 × 4 intra prediction modes[1]

As an alternative to the 4 × 4 luma modes, the entire 16 × 16 luma component of a macroblock may be predicted in one operation. Four modes are available.(Figure 5)

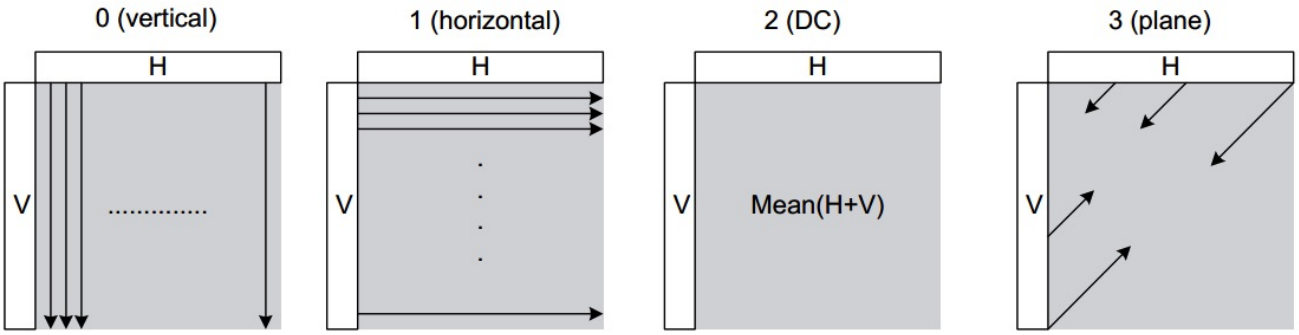

Figure 5. 16 × 16 intra prediction modes[1]

Intra prediction of the luma component with an 8 × 8 block size is only available in the High profiles. Each 8 × 8 luma block in a macroblock is predicted using one of nine prediction modes which are as same as the nine modes of 4 × 4 luma.

Each chroma component of a macroblock is predicted from previously encoded chroma samples above and/or to the left, with both chroma components always using the same prediction mode. The four prediction modes are similar to the 16 16 luma prediction modes. The numbering of the modes is different, but the rules are in common.

3.3.3 Inter-Prediction

Inter prediction is the process of predicting a block of luma and chroma samples from a reference picture that has previously been coded and transmitted. It takes advantage of the fact that the content of a new frame in the video often has high correlation to the data in the previous frames.The offset between the position of the current partition and the prediction region in the reference picture is a motion vector. The motion vector may point to integer, half- or quarter-sample positions in the luma component of the reference picture.

The decoded pictures stored in the Decoded Picture Buffer (DPB), in which case they may be used as reference pictures for inter prediction.The pictures in the DPB are listed in a particular order, and the list can be classified into three different types which are described in Table 3.3.3.

List0 (P slice)	A single list of all the reference pictures. By default, the first picture in the List is the most recently decoded picture.
List0 (B slice)	A list of all the reference pictures. By default, the first picture in the List is the picture **before** the current picture in display order.
List1 (B slice)	A list of all the reference pictures. By default, the first picture in the List is the picture **after** the current picture in display order.

Table 3. Three types of DPB list

Each 16 × 16 P or B macroblock may be predicted using a range of block sizes. The macroblock is split into one, two or four macroblock partitions:(a) one 16 × 16 macroblock partition (b) two 8 × 16 macroblock partitions (c) two 16 × 8 macroblock partitions (d) four 8 × 8 macroblock partitions

If 8 × 8 partition size is chosen, then each 8 × 8 block of luma samples and associated chroma samples, a sub-macroblock, is split into one, two or four sub-macroblock partitions): one 8 × 8, two 4 × 8, two 8 × 4 or four 4 × 4 sub-MB partitions. This process is illustrated in Figure 6.

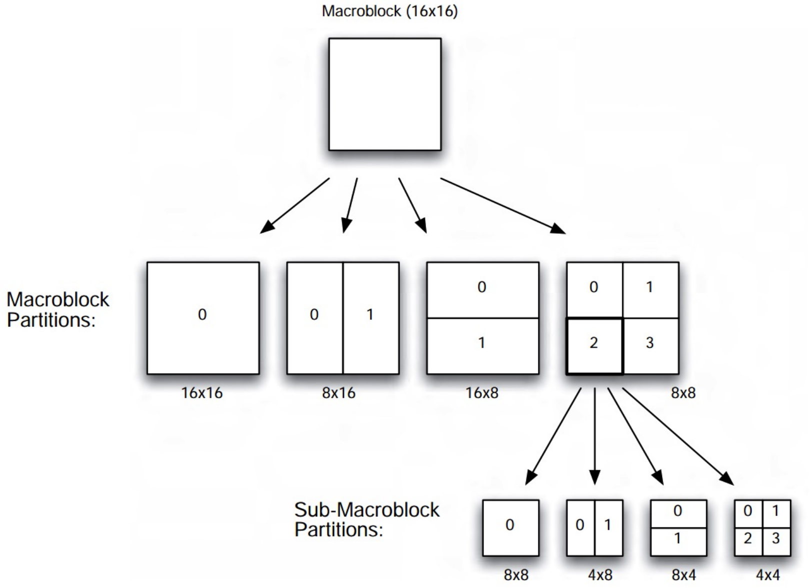

Figure 6. Macroblock partitions and sub-macroblock partitions[1]

Each macroblock partition and sub-macroblock partition has one or two motion vectors (x, y). Each of them points to an area of the same size in a reference frame that is used to predict the current partition. Motion vectors for neighboring partitions are often highly correlated and so each motion vector is predicted from vectors of nearby, previously coded partitions. Motion vector is supposed to compensate the decoded picture from the reference ones. It means we could transmit the motion vector instead of the whole redundant of reference marcoblock.

4. SOFTWARE DESIGN AND IMPLEMENTATION

The software architecture design is guided by the syntax of H.264 bitstream which is shown in Figure 7.

The project is implemented mainly in object oriented paradigm. Primary concepts of H.264 are abstracted as objects. Figure 8 is a simplified Unified Modeling Language(UML) diagram of the software architecture.

The functionalities of these objects are described here.

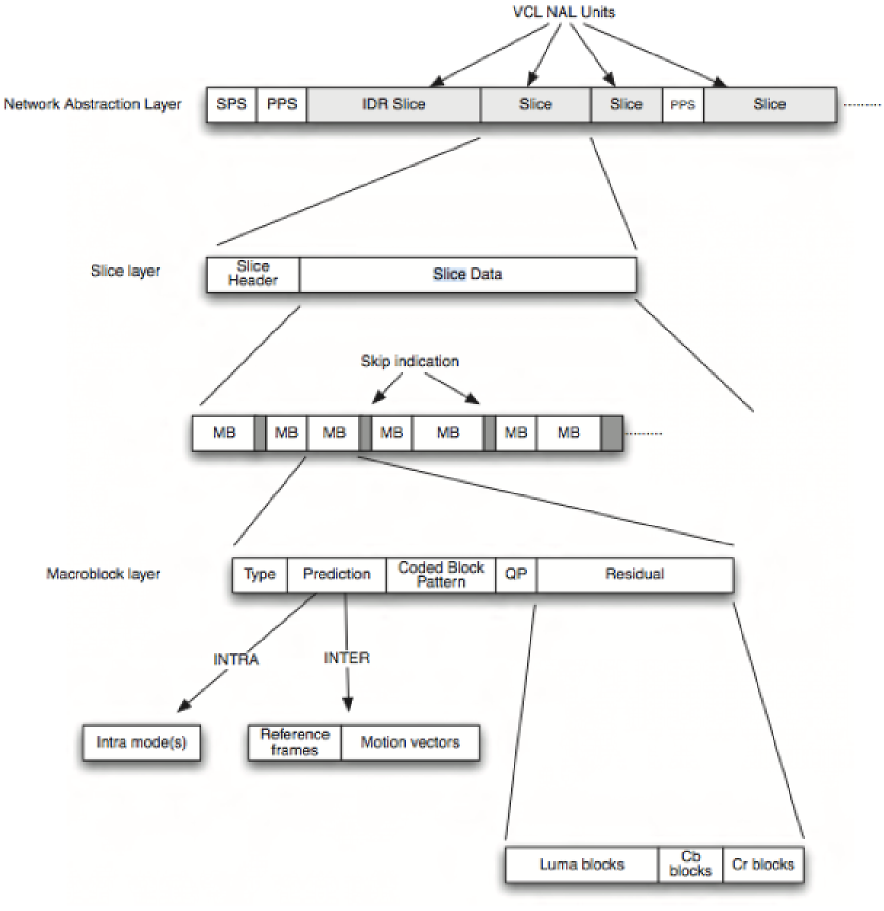

Figure 7. H.264 bitstream syntax[2]

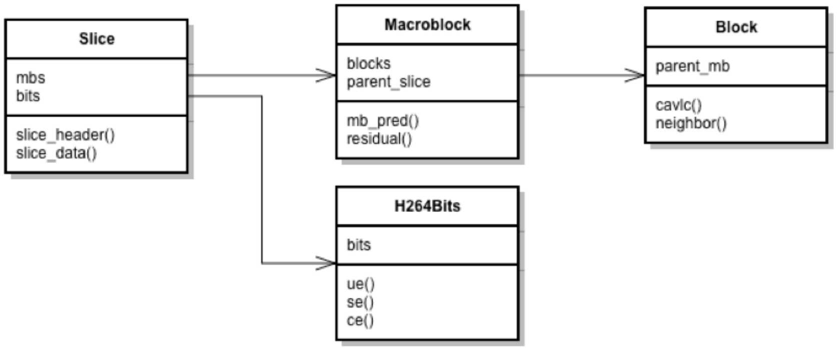

Figure 8. Software architecture UML diagram

4.1 H264Bits

H264Bits object stores raw bitstream data and entropy decoding methods. Each slice object contains a reference to the singleton H264Bits object, so the binary data can be decoded and transfered to its associated object.

4.2 SPS

SPS object contains sequence parameter set related parsing logic and result. The parsing methods read parameters from bitstream and store the results as instance variables of the object. So that these parameters can be shared later. There's usually only one SPS object in a sequence.

4.3 PPS

PPS object contains picture parameter set related parsing logic and result. This object has behavior similar to SPS except that there can be multiple PPS objects in a sequence.

4.4 Slice

Slice object contains slice header parsing logic, slice related parameters and macroblock objects. Slice-related parameters from slice header segments are parsed and stored here. Arbitrary Macroblock objects are contained in this object which is identical to H.264's layer logic.

4.5 Macroblock

Macroblock object contains macroblock parsing logic and Luma/Chroma block objects Macroblock-related parameters are decoded and CAVLC entropy decoding process is invoked here. The entropy decoded results are stored as Block objects inside this object.

4.6 Block

Block object contains Luma/Chroma block's transform coefficients and related decoding methods. The object also has block id related information stored in it.

Each object also contains its parent object pointer to share data between each other. So accessing sequence parameter set from a luma block object can be done like this:

```
one_luma_block.mb.slice.sps.some_parameter
```

There are another two modules for Intra frame decoding: idct and intra_pred. These modules are collections of functions for inverse DCT (and dequantization) and intra prediction.

4.7 IDCT

This module contains amount of functions to perform dequantization and inverse DCT on block coefficients levels. Related argument tables are also stored here as module attributes.

4.8 Intra Prediction

This module contains amount of functions to generate intra prediction for blocks.

5. RESULT AND VERIFICATION

The project now has the capability to decode YCbCr components of the first intra frame from raw H.264 bitstream, which means the following functionalities have been implemented:

- Bitstream parsing
- Intra prediction generation
- Inverse quantization
- Inverse transformation

Figure 9 shows the reference frame and decoded frame of test data Foreman QCIF.

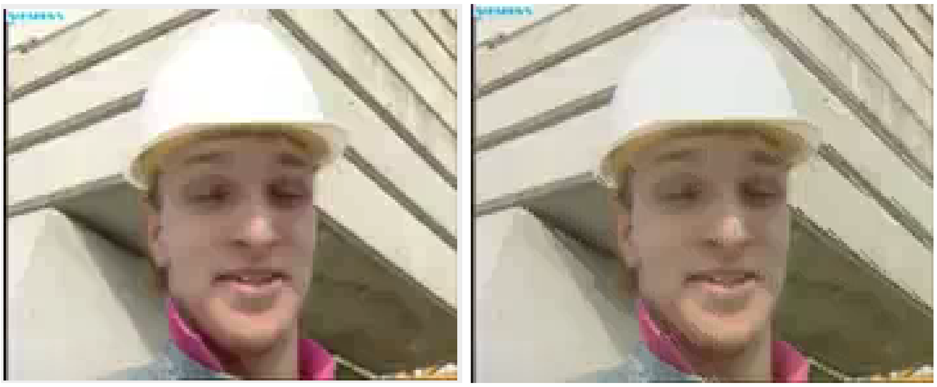

Figure 9. Reference frame (left) and decoded frame (right)

ACKNOWLEDGMENTS

We are thankful to Professor Jiying Zhao who provided expertise that greatly assisted the project.

REFERENCES

[1] Richardson, I. E., [*H. 264 and MPEG-4 video compression: video coding for next-generation multimedia*], John Wiley & Sons (2004).

[2] Team, J. V., "Advanced video coding for generic audiovisual services," *ITU-T Rec. H* **264**, 14496–10 (2003).

[3] Wiegand, T., Sullivan, G. J., Bjontegaard, G., and Luthra, A., "Overview of the h. 264/avc video coding standard," *IEEE Transactions on circuits and systems for video technology* **13**(7), 560–576 (2003).

[4] Schwarz, H., Marpe, D., and Wiegand, T., "Overview of the scalable video coding extension of the h. 264/avc standard," *IEEE Transactions on circuits and systems for video technology* **17**(9), 1103–1120 (2007).

[5] Horowitz, M., Joch, A., Kossentini, F., and Hallapuro, A., "H. 264/avc baseline profile decoder complexity analysis," *IEEE transactions on circuits and systems for video technology* **13**(7), 704–716 (2003).