Lossless JPEG Design Guide

Grant Brown

University of Utah, LNIS

7/22/2020

Contents

Overview

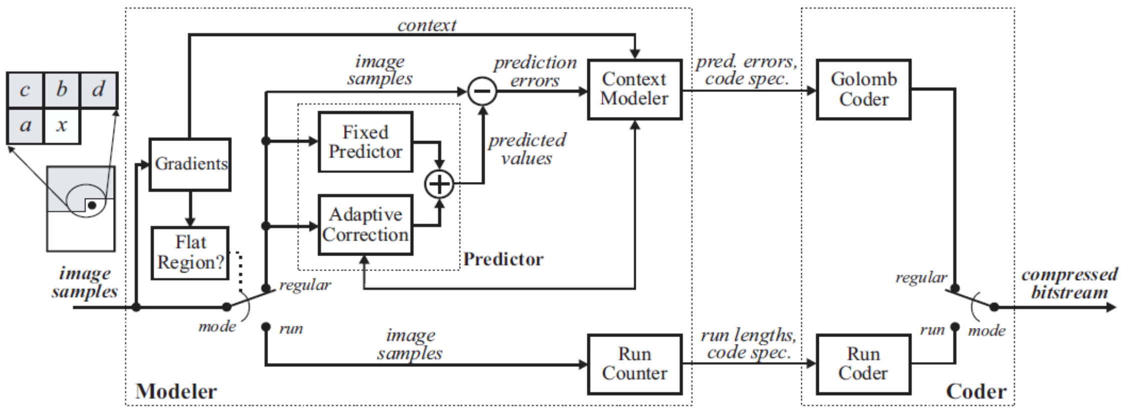


Figure 1: JPEG-LS: Block Diagram

The JPEG-LS implements lossless compression of continuous-tone images. The baseline for the JPEG-LS is given by the ITU-T Recommendation T.87.

The ISO/IEC International Standard:

- Specifies a process for converting image data to compressed binary data
- Specifies a process for converting the compressed binary data into reconstructed image data
- Specifies formatting for the sentinels of the process

The coding principle is defined by a source image being input pixel by pixel into the encoder in a raster scan pattern. Context Modeling is achieved after scanning past data, inferences can be made on the current sample by assigning a conditional probability to the value of the parameter. Using probability analysis predictions can be made to minimize the value of the encoded pixel so as to reduce the encoded length modulo ALPHA (context parameter). An in-depth overview of the JPEG-LS algorithm will not be restated in the design guide, any additional information can be addressed through reading working principle documents submitted along with the design guide. **It is imperative that the reader reads the ITU T.87 and HP JPEG attached papers before reading this guide.**

The JPEG-LS has been implemented in Verilog as a stall-less 7 stage pipeline design. Overview of the modules in each stage are:

Stage 1

- Get Data FSM

Stage 2

- Gradient Quantization
- Run Interruption Type Determination
- Mode Determination
- Run Counter

Stage 3

- Context Gradient
- Predictor Stage 1
- Run Coding

Stage 3.5

- Determine Context Read/Write

Stage 4

- N Update
- Context Multiplexer
- Predictor Stage 2
- Temp Calculation

Stage 5

- Context Update
- Error Modulo Mapping
- K Calculation
- Run Length Adjustment

Stage 6

- Rice Encoder

Stage 7

- Bit Packer

Hardware design contains feedback, which is needed to ensure the context modeling of the parameters are current to avoid stalling while encoding. Test input and output parameters are generated for the design using two MATLAB files, one for correct bitstream output, and one for collection of internal parameters. A python script is used to generate binary format for these numbers, and then $readmemb is used to read the data into testbenches, which compares actual output vs expected output. The reason behind two testbench scripts is one is designed like the hardware implementation so it allows us to obtain all the internal variables, and one is designed at a higher level in a software approach, which is used for easier understanding and is used to obtain the byte output of the design.

Stage 1

Get Data FSM

The Get Data FSM is the start of the JPEG-LS pipeline. The Mealy FSM is designed to iterate through an external memory, grabbing the current pixel along with context pixels located to the West, North-West, North, and North-East. A diagram depicting this is shown below:

Rc	*Rb*	*Rd*
Ra	*x*	

Special use cases for edges of frames are needed to be accounted for:

Pixel	Position	Exception	Value Assigned
Ra	west	col = 0 and row = 0	0
		col = 0 and row ≠ 0	*Rb*
Rb	north	row = 0	0
Rc	northwest	row = 0	0
		col = 0 and row ≠ 0	*Ra* from previous time col = 0
Rd	northeast	row = 0	0
		col = N_COLS-1 and row ≠ 0	*Rb*

Table 3.7. Context Assignment for Special Cases.

Overview of Implementation

Implementation for the context and current pixel design is driven through the use of internal registers. To get the FSM started, we will have a master FSM send a FSMStart flag to the slave FSM. This should be sent after the global reset for all memories and registers located in the JPEG-LS design. At the beginning of the encoding, the previous encoded row is all 0's so we only need to load the x pixel, with all other context registers being reset to 0. The first 3 pixels will need to be saved as previous context pixels (Rc,Rb,Rd) for the subsequent row (row 2). We will continue the encoding till the EOL, where we will load the previously saved context pixels (Rc,Rb,Rd,Ra) into the current context pixels. This ensures the pipeline will not have a windup every time we increment the row. The second row will still have the context pixel, Rc, be 0 due to the edge case at the first column, in which we use context pixel, Ra, from the previous row is 0 due to the first encoded row. Once we hit the third row then the context pixel, Rc, will start to affect the design at the edge case of the first column.

As we move through the rows, we will need obtain the current rows pixel along with the previous row's pixels. This requires the internal registers of column index and previous column index, which are used to map the current internal state to a memory address in the external pixel memory.

As for the encoding scheme, the frames of the data are appended onto each other so we will encode the full depth as one image, not treat each frame as a portion of an interleaving pattern via the use of sentinels to signify the start and end of frames. In addition, EOL is an important flag as if we are in run mode (mode == 1) it can be interrupted by the end of a line (mode == 3), which follows a different encoding pattern than run interruption mode (mode == 2). Therefore, this FSM not only provides the pixels but drives the behavior of the design via flags. Once the FSM reaches the EOF flag we will pass this through to the Bit Packer FSM at the final stage to ensure the footer sentinel is sent to the decoder to signify the end of current image. The start_enc flag is used as a global enable signal to "power up" the design, and will be used to determine when data will be passed from one stage to another.

Mealy Logic Implementation

The Get Data FSM is a 3-stage FSM. The stages are IDLE, GET_DATA_ROW_ONE, GET_DATA.

IDLE – The IDLE stage is the waiting stage the FSM will be in until it receives an external start signal. When the FSM obtains this start signal it will trigger a read to the dual port RF Port A (read port) which will read the first pixel of the first pixel memory and load it to the FSM. The FSM will then increment the column number and set an internal flag, FSM_started, which moves the FSM to the second condition in the IDLE stage. On the second condition the pixel has been read and is currently inputted to the FSM. It will take the pixelIn value and store it to the current pixel in the encoding process, x. On this second condition the FSM will also read the second pixel in the memory and increment the column to get ready to read the third pixel. After accomplishing the second condition in IDLE the state will be incremented to the GET_DATA_ROW_ONE state.

GET_DATA_ROW_ONE – The GET_DATA_ROW_ONE stage is responsible for grabbing the correct context pixels for the first row. Since the values of the previous row pixels are all 0 this stage will only need to read the current row pixels. The IDLE stage does the instantiation and gets the FSM ready to start when it moves to this stage. The first pixel has been saved, the second pixel has been read, and the third pixel is ready to be read with the correct column number. On the first iteration in this stage we will save the current pixel to the previous B context pixel register, along with read the third pixel, save the second pixel to Rx, shift the current saved pixel to Ra, and send the start_enc flag to enable the second stage pipeline registers. On the next iteration we will need to save the current pixel to the previous Rd context pixel, along with read the next pixel, save the currently inputted pixel, shift the Rx pixel to Ra, and enable the second stage pipeline to encode the currently saved pixel. The FSM will follow this shifting sequence until it encodes all pixels on the current column, which the FSM will switch to loading the previous context pixel values to the current context pixel values. This is extremely important as it removes any unneeded “wind-up” to the pipeline. At the last encoding step we will send the EOL flag which is used to stop any run encoding processes. After the encoding of this row, we will reset the column register, increment the row register, and move to the next stage of GET_DATA.

GET_DATA – The GET_DATA stage is very similar to the prior stage except there is an added complexity of needing to load the previous rows context pixels into the context pixel registers (Rb, Rc, Rd). In order to do this there is an added column register, prev_col_index, which is used to identify the current pixel to be read from the previous pixel memory. Once we start this stage the previous rows context pixels have already been loaded into internal registers, however the current rows pixels will need to be configured just as the previous stage, and the previous rows pixel will need to be read to get the shifting operation ready for Rc, Rb, and Rd. The shifting operation is straightforward, as we read the previous pixel row memory, we will need to shift the context pixel Rd to Rb, Rb to Rc, and load the read pixel to Rd (if confused refer to the context pixel mapping diagram above). Therefore, we will enter the process of reading the first pixel, incrementing the column register, and reading the previous rows pixel to get the shifting operation ready. We will not save this previous row pixel as the instantiation of the current row has not been completed and encoding has not begun for the current row. To complete the instantiation operation, we will save the currently read pixel, read the next pixel, and increment the column index register to be ready to read the 3rd pixel of the current row. This will conclude the instantiation operation and will allow us to proceed with the current row encoding. At the first step of the encoding (col_index == 2) we will save the currently saved pixel to the previous context pixel memory registers of Ra and Rb, read the next previous row context pixel, read current row context pixel, perform shifting, and save the currently loaded context pixel. On the next operation we will perform the same actions but save the currently saved pixel to the previous context pixel memory register of Rd. We will continue this until the end of the line in which the EOL flag is set and will stop any current run encoding processes. There will be an additional check to see if the current depth index is 239 (311 x 135 x 240) and if so, will send an EOF flag along with the EOL flag to trigger a later stage to send a footer sentinel.

To perform the shifting and saving operations of the context pixels of the previous row (Rb,Rc,Rd) a case statement is employed to depending on the value of the savePrevPixelContext flag. There are 3 separate stages to the case statement. Stage 2’b01 is used at the end of a row to load the previous context pixel registers to the current context pixel registers. Stage 2’b10 is used to perform the shifting operation. Stage 2’b11 to cover the edge case of the final column encoding as specified in the diagram above.

A separate shifting operation needs to be performed between context pixel registers Rx and Ra after the first column. A separate behavioral block is used with a separate flag savePixelContext is used to cover this case.

To control the external pixel memory, we will have two separate read_pixel and read_prev_pixel flags set by the FSM which control behavioral blocks. Once set, behavioral block will trigger and use the value of read_MEM or read_Prev_MEM to set either the read_MEM flags which are fed as enable signals to the external memory modules.

The use of two external pixel memories is a design choice I made, it adds complexity and timing synchronization with the external FSM loading that memory. We cannot start loading the previous rows pixel memory until after we have read that address position. Depending on design choice this may require an extra flag or modification of this Get Data FSM. Another solution is adding another external memory row, which would require the reading memory portion of this FSM to be modified to accommodate the extra memory. However, the behavior should still follow a cyclic pattern. I will leave that decision to you and your team.

Stage 2

Gradient Quantization

The context pixels will determine what context number we will map too. The desired low complexity of the JPEG-LS design is somewhat hindered through this large context pool. To combat this, the quantization of the context numbers are broken into regions of symmetric equiprobable quadrants indexed -T,…,-1,0,1,…,T. Therefore, the context pool covers a total of $(2T+1)^3$ different context numbers. For JPEG-LS, T = 4 was selected resulting in, 729 contexts before merging. However due to the symmetry of the TSGD mapping of the prediction we can perform tail-merging of the contexts, turning the context mapping into an even function, with C(D) = C(-D), resulting in 365 regular/run mode contexts.

The context mapping is broken into two parts to ensure a avoid a critical path in the pipeline. The gradients are determined from the differences:

$$D_1 = d - b$$

$$D_2 = b - c$$

$$D_3 = c - a$$

These gradients represent the smoothness and edginess of the surrounding pixels which is the driving factor into the statistics in error prediction The gradient calculation is a signed calculation, however as mentioned above due to the TSGD merging we will need to determine the sign of the context, which is used in prediction analysis. Therefore, we break this design into 3 separate modules:

1) Gradient Calculation – Performs the simple discrete gradient calculation
2) Context Sign – Determines the sign of the context, sets the sign flag
3) Gradient Quantization – Maps signs of each gradient too the context indices

Gradient Quantization does not do the tail merging in this stage. The process follows Code segment A.4 from ITU T.87:

Code segment A.4 – Quantization of the gradients

```
if (Di <= –T3) Qi = –4;
else if (Di <= –T2) Qi = –3;
else if (Di <= –T1) Qi = –2;
else if (Di < – NEAR) Qi = –1;
else if (Di <=  NEAR) Qi = 0;
else if (Di <  T1) Qi = 1;
else if (Di <  T2) Qi = 2;
else if (Di <  T3) Qi = 3;
else Qi = 4;
```

Implementation

A 2’s complement implementation is used, with 15 mapping to -1, 14 mapping to -2, 13 mapping to -3, and 12 mapping to -4. If the number is determined to be negative by checking the MSB then it is inverted to its positive 2’s complement representation which will do a check against the values listed above. Thresholds may be modified to meet user satisfaction, but the JPEG group has determined T3 = 21, T2 = 7 and T1 = 3 to be the given thresholds for the JPEG-LS encoding process. To avoid code overflow in this module, an internal module of ContextCalculation is used to implement the discrete gradient calculation.

This stage should output just the signed context indices, along with the context sign. The context sign is made to be implemented in a priority encoding manner. The first non-zero element in sequential order from Q1 to Q3 determines the sign. If the first non-zero element is positive, we map to a positive context and if negative we map the context negative context. To avoid code overflow in this module, an internal module ContextSign is used to implement this context sign determination process.

RIType

RIType is used to determine which context of run interruption we are using. The implementation is modeled after Code segment A.17 in ITU T.87:

Code segment A.17 – Index computation

```
if (abs(Ra – Rb) <= NEAR
        RItype = 1;
else
        RItype = 0;
```

Implementation

Implementation is solely based upon the context pixels of a/b. If a == b then the RIType == 1, else RIType == 0. For the latter case, later in the pipeline a context pixel comparison of a/b is needed to determine which number is greater. This is performed within this module as a comparison check is already being performed. This avoids excesses busses being fed through the pipeline.

Mode Determination

Mode Determination is used to determine the mode based on the current context pixels. Four modes can be achieved in this module:

0) Regular Mode - $D_1 \mathrel{!}= 0 \vee D_2 \mathrel{!}= 0 \vee D_3 \mathrel{!}= 0$
1) Run Mode - $D_1 == 0 \wedge D_2 == 0 \wedge D_3 == 0 \wedge\ x == a \wedge\ !EOL$
2) Run Interruption Mode - $D_1 == 0 \wedge D_2 == 0 \wedge D_3 == 0 \wedge x\ ==\ a \wedge !EOL$
3) EOL Interruption Mode - $D_1 == 0 \wedge D_2 == 0 \wedge D_3 == 0 \wedge x\ ==\ a \wedge EOL$

The mode creates a basis for residual prediction and Rice Encoding. EOL breaks the run mode and brings the same into a special case of run interruption mode, which does not require the current pixel to be encoded since it was not the cause of run interruption**. If previous mode was run mode than the context numbers do not determine if the run is broken.** To break the run mode the currently encoded pixel must be different than the run value pixel that started the run, which shows a deviation of the image from the current saturation.

Implementation

To determine which process must be employed for mode determination, an internal register called Previous_Mode is used. If the previous mode was previous_mode == 1, or run mode, then the considerations above are shown, in which run will be broken if EOL or if x != a, as a is the run value. In the case which the previous mode was a non-run mode, or previous_mode != 1, we will use the conditions shown in the figure above to determine the mode for encoding.

Run Counter

The Run Counter module keeps track of the current run count for a run mode encoding process. If the run mode is interrupted by the EOL flag, then the run count will need to be reset to 0 for the subsequent sample. The run count cannot be reset immediately due to the need of the current run count to be encoded based on the value of J[Run Index]. Therefore, the previous mode is kept internal and an internal run counter wire is set as a multiplexer depending on the previous mode. In run interruption mode, we can reset the value of the current run to 0 as the J(Run Index) bits of run count have been encoded on the last run mode pixel. This is due to forwarding in the pipeline.

Implementation

Implementation of this module is fairly straightforward. The run value is always assigned to x, as if the run is current the run value will be associated to x. An internal variable called run_count_current is used to determine what the run count is. If the previous mode was a non EOL run interruption mode then we will continue coding like normal. However, if the previous mode was an EOL run interruption mode the run count was needing to be encoded at the EOL run interruption so we could not reset it immediately. The immediate mode after the EOL run interruption may be a run mode and will need a reset run count to increment upon however. To deal with this an internal check is used to see what the previous mode is, in which it is reset under the correct conditions. The rest of the module just shows incrementing when conditions are met, otherwise the run count is reset to 0.

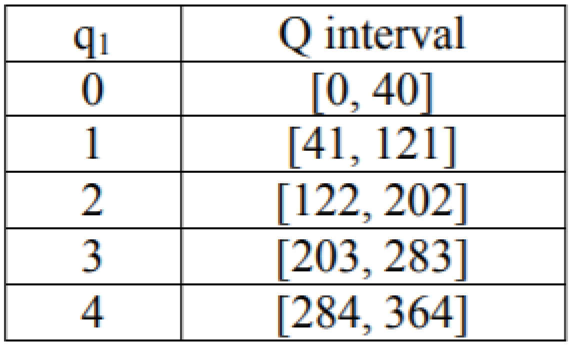
Stage 3

Context Gradient

Stage 2 of the context number determination. This stage will take the three quantized signed context numbers outputted from the Gradient Quantization module and will perform a mapping of the three quantized context numbers to a single unsigned context number, Q. The ITU T.87 does not specify a standard for the mapping of <Q1, Q2, Q3>, however it does note that every possible context must be an injective function. To accomplish this the equation used is:

$$Q = 81q_1 + 9q_2 + q_3$$

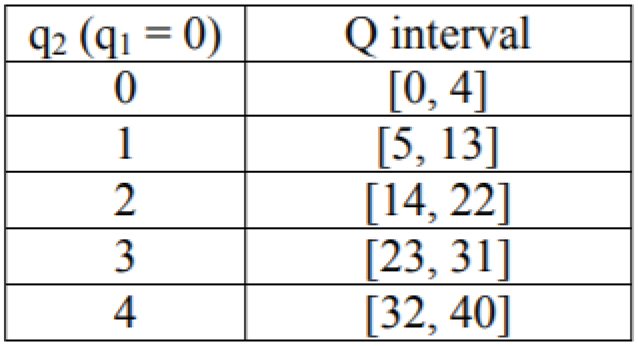
This equation gives the mapping of q_1 as follows:



q_1	Q interval
0	[0, 40]
1	[41, 121]
2	[122, 202]
3	[203, 283]
4	[284, 364]

This interval account for all possible context outcomes depending on the case of q_1.

If $q_1 = 0$, we will get the possible mappings of Q based of q_2 as follows:



q_2 ($q_1 = 0$)	Q interval
0	[0, 4]
1	[5, 13]
2	[14, 22]
3	[23, 31]
4	[32, 40]

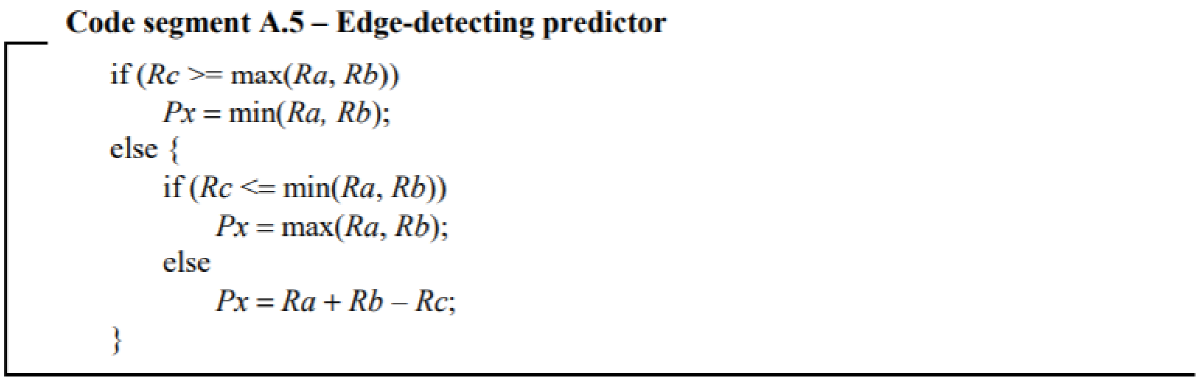
If $q_2 = 0$, then q_3 will result in Q mapping to [0,4].

Implementation

This module will break the equation for Q into three separate iterations in in which a check is done on the sign of the specified context number and a sequence of multiplication and addition will combine the numbers together. Sign extension is used due to the quantized sign numbers being negative. This allows signed multiplication to follow regular partial product algorithms. Special cases are met for the run interruption mode, in which depending on the RIType value the context number will be either 366 or 367 (355 or 366 due to 0 indexing). If mode is run mode the context number will be mapped to 0.

Predictor

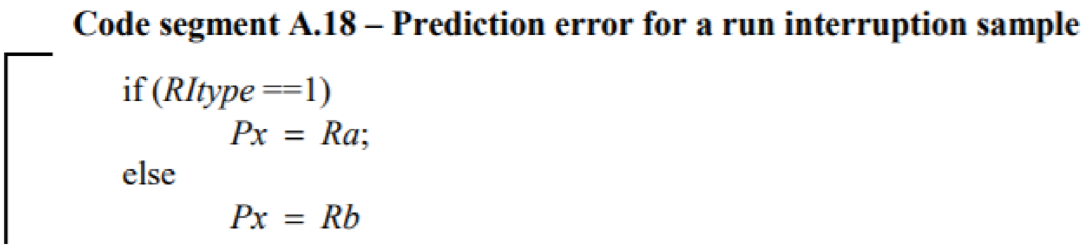
This module covers the first stage of prediction based on context pixel values as described in Code segment A.5 for regular mode encoding:



Code segment A.5 – Edge-detecting predictor

```
if (Rc >= max(Ra, Rb))
    Px = min(Ra, Rb);
else {
    if (Rc <= min(Ra, Rb))
        Px = max(Ra, Rb);
    else
        Px = Ra + Rb – Rc;
}
```

Or Code segment A.18 in run interruption coding:



Code segment A.18 – Prediction error for a run interruption sample

```
if (RItype ==1)
        Px = Ra;
else
        Px = Rb
```

The fixed predictor performs a test to detect vertical or horizontal edges. It will tend to pick the context pixel b when there is a vertical left edge to the current pixel, the context pixel a when there is a horizontal edge above the current pixel, or a median of context pixels a, b, and x if no edge is detected. The principle is to keep a hyperplane between the 4 pixels, determining a smooth image. Due to the principle of design, the predictor is termed as a “median edge predictor” (MED). Therefore, the quantitative mapping is:

$$\hat{x}_{\mathrm{MED}} \triangleq \begin{cases} \min(a,\, b) & \text{if } c \geq \max(a,\, b) \\ \max(a,\, b) & \text{if } c \leq \min(a,\, b) \\ a + b - c & \text{otherwise.} \end{cases}$$

In the case where we are entered into a run interruption mode, the context pixels of a and b along with RIType will determine the predicted value. EOL Interruption and run mode do not use this module.

Implementation

The mode calculated in the ModeDetermination module is fed into this module to determine which code segment to follow from the ITU T.87. If the mode is regular mode, then the quantitative mapping above is performed. An internal comparator will compare the a and b values, and will assign the minimum and maximum value to the respective internal busses. These busses are used in the behavioral block to perform the comparison checks needed for the MED predictor. In the run interruption case the RIType value, which is calculated in the RIType module, is fed into this module to determine whether a or b will be the predicted value. Due to EOL run interruption mode and run mode not using this module, the predicted value is kept 0 in this case to avoid dynamic power dissipation.

Run Coder

The Run Coder is one of the most complex modules due to the variation and casual nature built into the algorithm. The principle based behind this module is it needs to provide the context values of J[Run Index], Run Count Compare, and Run Index for the run interruption mode coding process. In software design, this is done all at the end of a run sequence, with an accumulated run count which is iterates through the J[Run Index] memory and compares it to run count as in Code segment A.15:

Code segment A.15 – Encoding of run segments of length *rg*

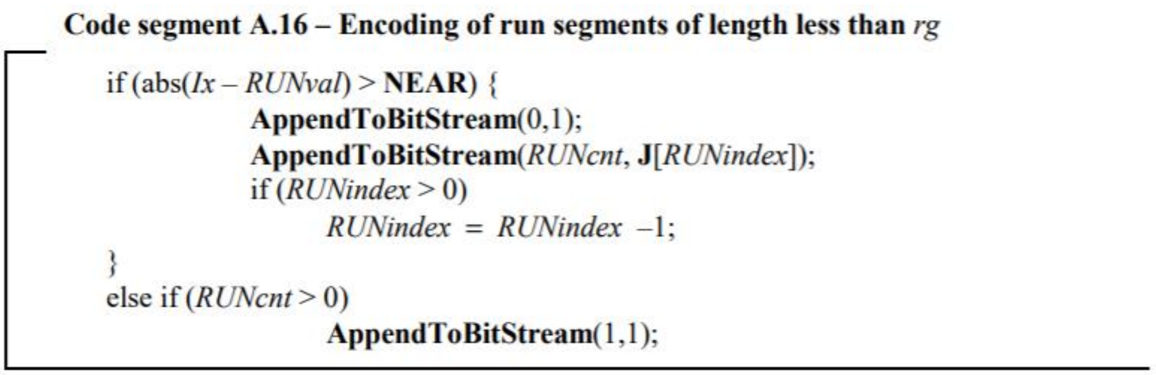
```
while (RUNcnt >= (1 << J[RUNindex]) ) {
        AppendToBitStream(1,1);
        RUNcnt = RUNcnt – (1 << J[RUNindex]);
        if (RUNindex < 31)
                RUNindex = RUNindex +1;
}
```

However, in hardware this would cause a massive (depending on the run length and number of hits) stall. To avoid this, I designed a casual system which encodes the run as it is in progress.

The principle behind what I call “hit encoding” is based upon the 32 indices in the J Register File. J is composed of 32 stored values as:

J = [0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3,4,4,5,5,6,6,7,7,8,9,10,11,12,13,14,15]

Each time a run count equals 1 << J[Run Index] we accumulated a “hit”, in which we need to append a ‘1’ onto the bitstream. In order to do this during the actual run mode encoding process, I keep track of the current run count compare values as associated with the value of 1 << J[Run Index], and whenever it hits this value I set a flag which is passed down the pipeline to append a ‘1’. If the run continues we need to determine what the next “hit” value is. This is accomplished by performing an accumulate function on the run count and 1 << J[Run Index + 1]. In addition, there are a multitude of special and edge cases of indices that need to be accounted for. J[Run Index] needs to be preserved for the current run interruption and run coding to be able to perform Code segment A.16:



Code segment A.16 – Encoding of run segments of length less than *rg*

```
if (abs(Ix – RUNval) > NEAR) {
        AppendToBitStream(0,1);
        AppendToBitStream(RUNcnt, J[RUNindex]);
        if (RUNindex > 0)
            RUNindex = RUNindex –1;
}
else if (RUNcnt > 0)
            AppendToBitStream(1,1);
```

To account for this, we cannot decrement the Run Index until after the run interruption mode passes. However, after run interruption we can either go straight back into a run mode or into another run interruption mode. In the first case, the Run Index needs to show updated for the immediate next stage but cannot be decremented in the previous stage due to J[Run Index] needed for run interruption coding. To deal with this, internal J[Run Index]’s along with Run Index’s are kept separate from the global J[Run Index] and Run Index. If mode and previous mode meet certain conditions, we will interpret the design using different parameters:

Mode == 1 or Mode == 3 and Previous_Mode != 2 and Previous_Mode != 3: This is normal use case. No specific cases need to be accounted for here.

Previous_Mode == 2: As mentioned above if previous mode is 2 then we need to use the decremented Run Index value and the corresponding J[Run Index] value. The global J value and Run Index value cannot be used due to the need for them to be used in encoding processes, so they have not been updated to account for the decrement of the Run Index value. So, to combat this, a separate set of values called run_index_decision, run_count_compare_decision are used. These values are evaluated in combinational continuous assignments that use mode and previous_mode to determine what their values are. They are decremented when the run interruption mode is occurring and stored in registers, which essentially hides the delay within the stage of the pipeline. In the beginning of the behavioral block, there is an update to the J[Run Index] if it meets certain conditions and an edge case, such as switching from index 4 to 5, in which J[Run Index] switched from 0 to 1 and run_count_compare becomes (1 << 0) or (1<<1).

If mode == 2 and previous mode == 2 we need to undo the increment done in the combinational statement to the current Run Index Decision New value. However, it is still decremented for the next mode after the 2nd mode == 2.

Other conditions don't matter such as mode == 0 so we will pass the decremented values off to the global values and no other sequential updates to the Run Coder internal parameters happens in the immediate mode after the run interruption.

Prevous_Mode == 3 AND Mode == 1: In this case of EOL run interruption we do not have any decrementing happening on run index or J. J_Comp is the saved value from the last J[Run Index] so we use that as a run count compare placeholder. We cannot use J[Run Index] since if it is an edge case J[Run Index] could have been incremented in anticipation for the next run count compare value.

Prevous_Mode == 3 AND Mode == 0: We only need to update the global variable with the new values. Run index is already set to the correct value due to no incrementing, but run_counter_compare needs to be reset if it was an accumulated value.

Run count compare may be accumulated if mode == 2, since run count is accumulating and we want to determine what run count value is the next hit. When the run is interrupted we need to reset the accumulated run count value to the value associated with the 1 << J(Run Index). To do this and count for edge cases in the change of J[Run Index] values an internal combinational block uses conditional flags to determine if J_run_count_compare should mirror J_Comp, in which J was not an edge case after Run Index was decremented, or if J_run_count_compare needs to be J_Comp – 1, in which an edge cases has been encountered. That is what these conditions mirror. Additionally, for use in encoding, a few internal variables such as J_Comp and J_Recurring_Mode_Two are passed down the pipeline and used by the Rice Encoder.

To provide the J[Run Index] values of [0,…,15] an internal Register File was created which takes a run_index_reset_compare value to determine which J register should be output. It should mirror a multiplexer output. Run_index_reset_compare follows the same decrementing algorithm as shown above.

Stage 3.5

Stage 3 Registers

There is a special comparison being made in the Stage 3 Regiters modules that enables the stallless nature of run encoding according to Code segment A.16 to be performed. Forwarding in the pipeline is employed to determine what the next mode is. If we are in run mode and the next mode is in run interruption mode, a do_run_encoding flag is set to signify the Rice Encoder needs to perform Code segment A.16 along with normal run hit encoding. If the current mode is 2 and the previous mode was a non_run mode, then we will need to perform Code segment A.16 in the run interruption encoding sample. The do_run_encoding flag is used to signify this as well.

Determine Context Read/Write

This module is solely responsible for determining when the feedback values from Context Update module need to be sent to memory, and when the context values associated with a specific context number need to be read. Context values such as A, B, C, N, and Nn will always need to be read and written to memory on the same cycle unless the feedback context number equals the current calculated context number, in which the feedback values are just used immediately. We will still be writing the context values to memory along with using them in subsequent analysis. The feedback values are just the previously read context values associated with a specific context number. After performing encoding with these context values, they will be updated and feedback to this module. Context variables are not used in run encoding processes, therefore read and write will not affect the context number Q.

Implementation

The useFeedbackValuesDecision value determines whether or not to use the feedback values associated with the Context Update module. This module is two stages away (two clock cycles), therefore we must have a check for whether or not Q == Q_Feedback along with a counter of start_enc_count > 2. This count is needed to ensure that it has been two clock cycles since the first start_enc signal has been received by the module.

Two internal values are kept by the module to determine read and write conditions. The values are read_Context_Memory and write_Context_Memory. The read_Context_Memory must be fed to each of the individual dual port RF memories. It will trigger a read corresponding to the Q_Read value. The Q_Read value will be used as the address indicator for the memory. The check looks to see if the Q_Feedback value equals the currently calculated context mapped Q value and determines if a read is needed in this occasion. Unlike the read operation, context variable updadted values will always need to be written to memory regardless of the feedback comparison check. Therefore, a start_enc_feedback variable is fed back from the Context Update module to this module to enable a write context memory operation according to the Q_Write variable. Q_Write works just as Q_Read does and will be fed to the context memory dual port RFs as the address indicator.

Internal registers are used to save the feedback values when Q_Feedback == Q. These values are outputted from this module to the Stage 4 Context Mux module, which will determine which context variable values to pass through for subsequent analysis.

To write to the external context memories, a value being output from the JPEG module, write_Context_Memory, will be either one of two values. This value is assigned from the determineWrite variable, which is determined in the Context_Update module later in the pipeline. If the mode was regular mode, then the write_Context_Memory will need to write the updated values of A, B, C, and N to the respestive context memories. If the mode was a run interruption mode, then the write_Context_Memory will need to write the updated values of A, N, and Nn to the respective memories. In the external memory modules, a comparison will be used to check for these values of write_Context_Memory. If the value is equal to either 1 or 2, then the enable signals to the respective memories will need to be triggered so as to create a write condition on the next posedge clk.

Stage 4

N Update

Updates N counter context variable by 1. If N is equal to a threshold set by JPEG-LS (N == 64), then a reset flag will be set to divide the context variables by 2.

Implementation

N is a 7-bit unsigned value, therefore a shift left operation will be used as the division operation. A binary ‘1’ is added regardless of the threshold comparison to increment the N value. A flag, resetFlag, is used to indicate that the other context variables will need to be updated in accordance to Code segment A.23:

```
if (N[Q] == RESET) {
            A[Q]  =  A[Q] >> 1;
            N[Q]  =  N[Q] >> 1;
            Nn[Q]  =  Nn[Q] >> 1;
}
```

Or Code segment A.12:

```
if (N[Q] == RESET) {
        A[Q] == A[Q] >> 1;
        if (B[Q] >= 0)
                B[Q]  =  B[Q] >> 1;
        else
                B[Q]  =  -((1-B[Q]) >> 1);
        N[Q]  =  N[Q] >> 1;
```

These will correspond to different encoding modes as there are different context variables used depending on the mode of encoding.

Context Mux

Multiplexer that determines which context variables are used by the design. Feedback values are used if the current Q value equals the one of the next two context numbers. The need for two conditional checks is if the same context number was calculated twice. We will need the updated context values immediately, however, the DetermineContextRead_Write module will not have detected this case as it is two pipeline stages behind. Therefore, this module will check this edge case to ensure correct encoding. Otherwise, the currently read context values are used.

Implementation

This module is very straightforward. It just uses flags and conditional checks to determine which values are used. The feedback values are saved in the DetermineContextRead_Write modules and fed into this module for use if needed.

Prediction Residual

Second stage of the MED predictor takes into account prediction error of the TSGD. This module is used to accomplished Code segment A.6 and Code segment A.7 of the ITU T.87 when in regular encoding:

Code segment A.6 – Prediction correction from the bias

```
if ($SIGN == +1$)
    $Px = Px + C[Q]$;
else
    $Px = Px - C[Q]$;
if ($Px >$ MAXVAL)
    $Px =$ MAXVAL;
else if ($Px < 0$)
    $Px = 0$;
```

Code segment A.7 – Computation of prediction error

```
$Errval = Ix - Px$;
if (SIGN $== -1$)
    $Errval = -Errval$;
```

In addition, it will be used to perform error value computation of Code segment A.18 and partial encoding of Code segment A.19 in run interruption encoding mode:

```
$Errval = Ix - Px$;
```

```
if (($RItype == 0$) && ($Ra > Rb$)) {
    $Errval = -Errval$;
    $SIGN = -1$;
}
else
    $SIGN = 1$;
```

Using the accumulated context variable, C, we will update the base prediction value by add this to the base prediction value outputted by the first predictor module to remove any existing bias. Then we need to clip the prediction value to keep it within a range for optimal coding. The range covers from 0 to 255 (8-bit prediction value). Then we will calculate the residual of the actual pixel, x, and the post-bias updated and clipped predicted value of x. If the context is negative, we then need to perform a 2's complement operation on the error value, otherwise the post-subtracted residual value is the final residual.

For Run Interruption mode we will simply perform a residual calculation with no bias updating. Depending on the value of the run interruption type and values of context pixels a and b, we may perform a 2’s complement operation, or multiplication by -1. In order to avoid excess busses in the design, the earlier RIType stage will do the context pixel comparison, and simply pass a flag into this module, “a_b_compare”, to perform this flag check.

EOL run interruption and run mode do not use the residual value in encoding processes.

Implementation

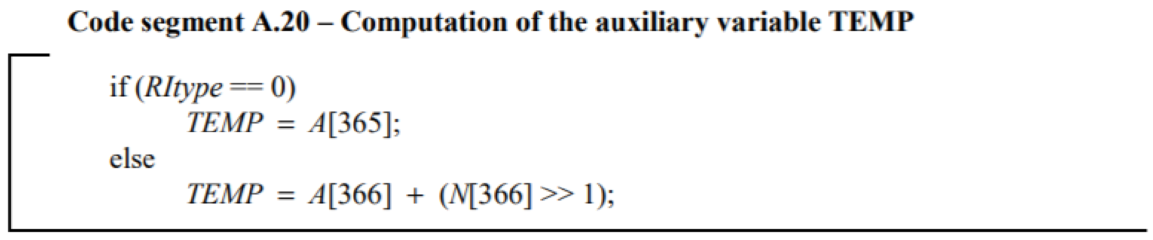
The output residual is determined by the mode input value. If the mode is regular mode encoding then first, we will perform the bias prediction update. This is assigned to an internal bus wire and then if the MSB of the wire is active high, corresponding to a negative value, the residual value is clipped to 0. Otherwise, if the value is greater than the range it will be clipped to 255. Afterword’s, the module will compute the residual value, followed by the sign check.

In run interruption mode coding the residual is calculated and the values of RIType and a_b_compare, which is the conditional check between the pixels a and b determined in the RIType module, determine if the residual needs to be inverted or not.

Run mode and EOL interruption mode do not use this mode so the residual is set to 0 to avoid dynamic power dissipation.

Temp Calculation

Temp is a variable that is used in run interruption mode Golomb variable k determination. According to ITU T.87, the variable TEMP will be replacing the accumulated context value A[Q] in the determination of k. Depending on the RIType value, TEMP will be calculated to either A[Q] exactly, or A[Q] + N[Q] / 2 rounded down to the floor. The algorithm for the computation of TEMP is displayed as Code Segment A.20 in ITU T.87:



Code segment A.20 – Computation of the auxiliary variable TEMP

```
if (RItype == 0)
        TEMP = A[365];
else
        TEMP = A[366] + (N[366] >> 1);
```

Implementation

This module is straightforward. N is an unsigned value so a shift right performs the division operation. The correct A is fed into the module by external means, therefore there is no concern in regards to having the correct context value, A.

Stage 5

Context Update

The prior knowledge given by the context variables gives insight into the structure of images. It allows this information to be compressed into context parameters which displays high-order dependencies of the images on predictive modeling mapping. The TSGD model for mapping as mentioned in HPL-98, contains a predictive error with a two-sided exponential decay rate. There is an associated DC offset to the predictive error signals due to integer constraints. To do adaptive correction of this bias we must keep a registry of context variables that associates error with occurrence.

Context Update performs the update of context variables for a specific context number to maintain these variables so as to create an efficient MED predictor. The context variables are as follows:

A - Magnitude of total modulo reduced error values. The max number of error values that can be accumulated is 64, and the max error is 128 (absolute value of -128) after modulo reduction. This shows that the A variable needs to be represented with 13 bits to meet the 8192 requirement.

B - Accumulative prediction residual. This is a signed value which is clamped during the bias computation module to ensure that the low complexity memory requirement is met for the context design. B is clamped to a value between -63 to 0. This results in a 7-bit value being needed.

C - Correction value which corrects the bias associated with a MED predicted value. It is seen as an average of fixed prediction errors. This means that it will be a signed number. C will maintain a value between 127 and -128 which is consistently incremented or decremented based on the current prediction error. This shows that the value C can be represented with 8 bits.

N – Provides a count on the number of times a context has been encountered. The max value is 64 before it is halved by the threshold comparison. Therefore, a 7-bit value is used.

Nn - Provides the negative threshold counter on run interruption coding. The maximum counter is 64 just as with N, therefore a 7-bit value can be used.

If the context was chosen it may introduce bias at a later date so bias cancellation values are calculated (C). Values such as N and A are used in prediction residual and Golomb coding calculates so a precise definition of context values is needed for accurate encoding.

The updating of the context variables must follow Code segment A.12 and Code segment A.13 for the regular mode encoding:

Code segment A.12 – Variables update

```
B[Q] = B[Q] + Errval *(2 *NEAR + 1);
A[Q] = A[Q] + abs(Errval);
if (N[Q] == RESET) {
        A[Q] == A[Q] >> 1;
        if (B[Q] >= 0)
                B[Q] = B[Q] >> 1;
        else
                B[Q] = –((1-B[Q]) >> 1);
        N[Q] = N[Q] >> 1;
}
  N[Q] = N[Q] + 1;
```

Code segment A.13 – Update of bias-related variables *B*[*Q*] and C[*Q*]

```
if (B[Q] <= –N[Q]) {
            B[Q] = B[Q] + N[Q];
            if (C[Q] > MIN_C)
               C[Q] = C[Q – 1;
            if (B[Q] <= –N[Q])
               B[Q] = –N[Q] + 1;
}
else if (B[Q] > 0) {
            B[Q] = B[Q] – N[Q];
            if C[Q] < MAX_C)
                    C[Q] = C[Q] + 1;
            if (B[Q] > 0)
                    B[Q] = 0
}
```

As shown, Nn is not used by this encoding mode. The constants MAX_C AND MIN_C are described in the ITU T.87 paper.

In run interruption mode the context variables change, with Nn, N, and A only being used. The update of context variables follows Code segment A.23 of the ITU T.87:

Code segment A.23 – Update of variables for run interruption sample

```
if Errval < 0)
        Nn[Q] = Nn[Q] + 1;
A[Q] = A[Q] + ((EMErrval + 1 RItype) >> 1);
if (N[Q] == RESET) {
            A[Q] = A[Q] >> 1;
            N[Q] = N[Q] >> 1;
            Nn[Q] = Nn[Q] >> 1;
}
N[Q] = N[Q] + 1;
```

Implementation

Context Update is broken into two modules. The top-level module is called ContextUpdate. The internal module called, BiasCancellation, performs the bulk of the computation needed for the updating process.

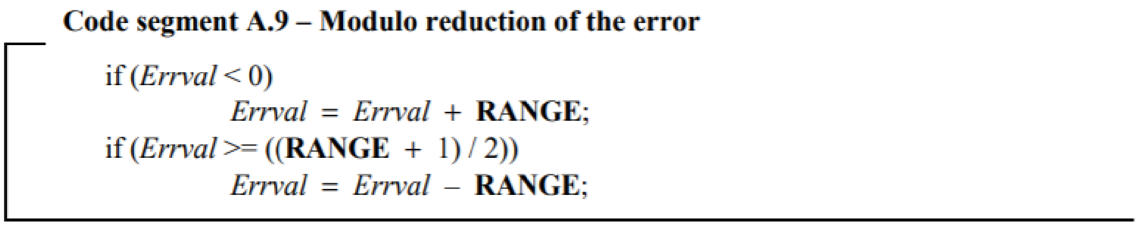
For regular encoding Bias Cancellation, the context variable update process starts with the beginning of Code segment A.12. Internal variables are used to store intermediate sums, and sign extension is needed to ensure correct arithmetic. If B < 0 the complex updating process of B is performed by individual continuous assignments in three separate parts. The sum is assigned to B_After_Flag if the value meets those conditions. To do the comparison check at the beginning of Code segment A.13 between B and N, we first look to see if B is negative. If B is negative, and the absolute value of B is greater than or equal to the updated N value, then we will enter that if statement. In that If statement we will follow encoding processes as stated in ITU T.87. C is checked against the min value. If C is positive then we are automatically greater than the min value. If C is negative, then viewed as an unsigned value, the more negative a number is the less the unsigned value is (up to the threshold mark). This is used in the comparison check. The B and N check is performed again with the updated value of B. The comparison check mirrors the process of the first check. If the first if statement comparison of B and N does not meet the conditions specified, then a second comparison check, which ensures that B is positive and greater than 0 is performed. The contents of the if statement then mirror Code segment A.13 exactly. If neither of these conditions are met then B is already between 0 and -63, and the B_After_Flag is mapped to the B_new variable, which is outputted as the feedback value.

In run interruption encoding Bias Cancellation, the context variable process is much simpler. If the error value is negative, which is determined by checking the MSB, then the Nn value is updated. A is updated according to Code segment A.23, and the reset is performed based off the reset flag generated by the NUpdater module.

If mode is EOL run interruption or run mode then the values are not adjusted.

Error Modulo Mapping

This module is adhering to coding segment A.9 of the ITU T.87 paper:



Code segment A.9 – Modulo reduction of the error

```
if (Errval < 0)
        Errval = Errval + RANGE;
if (Errval >= ((RANGE + 1) / 2))
        Errval = Errval – RANGE;
```

After modulo reduction and before passing to Rice Encoding, we must reduce the error residual to between (-ALPHA/2 < error_residual < Ceiling (Range/2) – 1). The error is reduced modulo alpha to maintain a standard range through the error residual adjustments. If the value is negative then we will add the alpha value to the residual to maintain the range relevant for coding. If the error residual is greater than half_alpha at this point then we will reduce the error value by alpha to move the value to the relevant range of coding. This is considered a modulo reduction and is passed through as the error residual value to the later stages in the pipeline.

Implementation

This module simply checks to see if the error value is negative by a MSB check, and if so it will add alpha or 256 to the error value. This may bring the error value above the positive threshold of 127, therefore another check needs to be made on the updated error value internal parameter. If this value is greater or equal to 128, then the value is decremented by 256, which ensures that it will be -128 or greater.

K Calculation

K calculation varies depending on the input mode. In run mode and EOL interruption mode k is not used during the encoding process so the only two modes of interest for k calculation are regular mode (mode == 0) and run interruption mode (mode == 2). The value of k is considered a Golomb (2-ary Rice) Encoding value, which determines how many bits of the MErrval value will be appended to the bitstream.

Regular Mode

During regular mode, the condition is based upon Coding segment A.10 of ITU T.87:

Code segment A.10 – Computation of the Golomg coding variable *k*

```
for(k=0; (N[Q]<<k)<A[Q]; k++);
```

Given the specific values of N and A a depending on the context number, we will increment k until the bitsll(N(Q), k) value is no longer less than the A value of the specified context. The incremented value will represent the k value which is passed onto Rice Encoding.

Implementation

Due to the nature of for loops not being synthesizable unless there is a given static increment range the k calculation module had to be given this condition. The max value of A is a 13-bit value and the minimum value of N is a 1-bit value. The greatest number of increments we could receive due to these conditions is 13. Therefore, implementation is based on a priority encoding scheme, in which comparators are built into the design that check the value of A against the shifted value of N. The comparator check will determine if the value of k is passed through with a 1 added to it or if the previous value of k is passed through without any addition performed. Using the max number of increments as an internal variable the comparator will be built into the for loop.

Run Interruption Mode

Run interruption mode follows the same principles as Regular Mode k determination, except that instead of doing our conditional check against A, we will introduce a new value, Temp, which was calculated during Stage 4. Temp is now used as the conditional check. We will use the same scheme of for loop with a predetermined range, but due to the addition of N, temp is now a 14 bit-value, so we will need to increase the number of incremental checks to 15.

Implementation

To reduce code complexity a module called K_Determination is built into the k_calculation module. This module implements the priority encoded scheme discussed above, with temp serving as the A conditional value in the run interruption mode.

Run Length Adjust

Run length adjust is used to create the remainder run length depending on our current run count and accumulated hits. Code Segment A.15 and A.16 describe this process, however this is all done in the background and this module is specifically performing this portion of the Code segment A.16 with passed in values already calculated:

[illegible]

$$RUNcnt = RUNcnt - (1 << \mathbf{J}[RUNindex]);$$

As mentioned before in run mode, I do a real-time encoding of the run counts. When it equals a run count compare value we append a 1 to the bitstream. However, when the run count is incrementing to the next run count compare value, there will be a differential between the run count and previous accumulated run count value aka previous run count compare value. If the run is interrupted before reaching the next hit or run count compare threshold, we will have a positive differential between these values. This differential is appended to the bitstream as (Run Count, J[Run Index]). This run count thou is not the accumulated run count, however just the differential, so this module calculates the differential. The previous run count compare value along with run count are passed to this module, so simply stated, this module is just a subtractor. This subtracted value is passed to the Rice Encoder as the total Run Count.

Stage 6

Rice Encoder

Rice encoder is a Golomb encoding scheme based on in which the divisor is a power of two. This module does the bulk of the work, in taking all of the conditional values, such as context numbers, error residuals, run counts, and performs a hardware mapped Golomb-Rice Coding on it. Depending on the mode being passed in, along with future modes (determined by previous stage(s) in the pipeline) we will perform different actions to encode values. The forwarding scheme, commonly used in pipeline-based schemas, thwarts stalls in the design. Due to the dependent and contrast behavior between the modes, individual modules were created for each mode, in which the encoding is done with a parallel fashion, and the output is set to the correct internal module output depending on the current mode. A key note made in the ITU T.87 shows that the maximum output is 32-bits, which points to limited-length, Golomb coding. We will proceed to discuss the different contexts we may receive and how encoding is proceeded:

Regular Mode:

Encoding regular mode follows the behavioral description of A.5.3 in ITU T.87. There are two conditions that may be met and we will take different behavior in each of these procedures:

1) If the number formed by the higher-order bits of MErrval is less than 23, this number stall be encoded onto the bitstream in unary zero-based representation, followed by a binary 1, and then the k-least significant bits of MErrval shall be appended to the bitstream.
2) If the number formed by the higher-order bits of MErrval is greater or equal to 23 then we will append 23 zeros to the bitstream, followed by a binary ‘1’. Then the binary representation of MErrval – 1 will be appended to the bitstream using 8 bits. With the MSB first followed by the decreasing order of bits.

Implementation

A design approach that is used in all modules is based upon a backwards approach to this problem. We will start with the end condition, appending to the LSB, and moving up towards the MSB as we append more bits depending on context values. For example, if condition 1 is met, we will start with appending the k-least significant bits of MErrval to the LSBs of the encoded_value variable, and working our way up to the MSBs with the binary ‘1’ and unary zero-based representation of the higher order bits of MErrval. All schemas in this module are based upon this backwards approach.

Due to the variable nature of the design, a for loop is not applicable in this case. In addition, due to the casual behavior of this design a priority encoded scheme has to be employed. This allows us to perform an action on the LSB, and the output of that will determine what happens to the neighboring bit. This priority nature is the core of the design and must be maintained.

To encode condition 1 an internal module is deployed which implements the priority based schema discussed above. The value of k is passed through the design, and conditional flags are used to determine if we need to keep appending the LSBs of MErrval, or to move on and append the ‘1’. Once the value of k is incremented to 0, we will have appended all of the MErrval LSBs for the current encoding scheme. We will then move onto appending the binary ‘1’ and the higher order bits are zero-based so will always be set to 0, this reduces complexity considerably and is a major reason in using the LSB based design. The value of MErrval is unsigned and will max out at 256, so it is represented by a 9-bit number. This shows that only 9-bits may be appended to the LSBs of the encoded_value. This is implemented by the design, with no further checks after the 8th conditional check of k.

Run Interruption Mode

This mode introduces extra complexity that is not taken on by the Regular Mode encoding. This is due edge cases, such as mode == 2 being passed through more than two times sequentially.

In normal run interruption operation, in which the previous mode was a run mode (previous mode == 1) we will perform an encoding scheme very similar to Regular encoding. A value called, EMErrval, is calculated as given in Code segment A.22 of ITU T.87:

Code segment A.22 – *Errval* mapping for run interruption sample

$$EMErrval = 2 * \mathrm{abs}(Errval) - RItype - map;$$

To ensure limited-length Golomb codes, if the limited code length is greater than 32 – J[Run Index] – 10 then we will perform the second condition as in regular mode encoding, and if less than we will perform the first procedure in the regular encoding scheme. Run Index is updated in the run interruption mode of the design by being incremented, but this Run Index value must represent the un-incremented value of Run Index, which introduces some complexity into the Run Coder module of Stage 3.

When we meet the condition of multiple run interruption modes being encoded sequentially (along with previous_mode == 0 in which there was no run buildup to the run interruption), we run into an added complexity. We will need to encode the normal operation as described above, but there was no previous run mode that was used to encode the Code segment A.16 in the ITU T.87. This shows we need to append a ‘0’ and J[Run Index] bits of Run Count to the bitstream. The trick of this schema is that Run Count will always be 0 during this case, so we won’t need to add any complexity to the hardware implementation, the upper bits will still be formed by 0 and we will just add this J value to the encoded_length so output compressor knows to read more bits from the output.

Implementation

The two separate encoding modules are employed in the RI mode are employed in a parallel fashion, which they are both calculated, and the correct output value is used depending on if an internal flag called “do_run_encoding” is set. Do_run_encoding, is generated internally in the Stage3Reigster module. It does a check on the current mode, and the next mode, which is stored in the previous stages pipeline register. This is an instance of forwarding in which we avoid stalls in the pipeline. The two modules mirror each other except for the fact that the run interruption coding of run count is appended to the bitstream of the sequential condition of multiple run interruptions.

Run Mode

Run Mode is a much simpler module. It either is encoding nothing when in run mode and the hit condition is not met, a ‘1’ to signify a hit according to Code segment A.15 in which run count == (1 << J[Run Index]) and we have not entered the last run mode, a ‘1’ to signify a hit and the ‘0’ and J[Run Index] bits of Run Count when we hit the last run mode, or just the ‘0’ and J[Run Index] bits of Run Count when we hit the last run mode. The last condition is inserted into the pipeline to ensure that we will not meet any stalling conditions. It uses the same forwarding scheme as discussed above with ‘do_run_encoding’.

Implementation

Implementation is once again broken up into a parallel scheme using internal modules, which will provide an output for each of the procedures and the output is chosen depending on the do_run_encoding and hit input flags. Do_run encoding is the flag that indicates the last run mode before run interruption, and the hit flag signifies the 1 appended to the bitstream when Run Count == (1 << J[Run Index]).

Stage 7

Bit Packer

The Bit Packer is a Mealy FSM that does the job of taking the encoded bit output and packing it into modulo byte-size output packages. The Bit Packer FSM will take the outputs of the Rice Encoder, and will compare the previously accumulated bits to the current input bits, and if the total accumulated bits is 8 or more, the module will package these modulo 8-bits into correct form and will output them to the bitstream. Many variations can happen in which 16, 24, 32, 40, 48, 56, or 64 bits will need to be outputted depending on the previous and current input size, making this a casual system. The total output size is 64 bits due to variations in inputs of the designs depending on edge cases, but the majority of the outputs will be of size 8.

The JPEG-LS compression stream requires starting and ending sentinel markers to tell the decoder when the current frame (or set of frames in this case) have been encoded. This Bit Packer FSM takes care of this job along with the bit packing. At the beginning of the JPEG-LS algorithm there will be a 6-stage delay in the pixels till the first potential output will need to be compressed for the bitstream. This allows the FSM to listen for the start signal (given by an external FSM) and begin outputting the header sentinel, which is stored internally as a register (localparam). It will take one clock cycles due to the header sentinel being 48-bits long, and then it will move to a waiting stage, in which it will wait for the start_enc signal which is sent by the FSM to trigger the start of the encoding process. The FSM will stay in this stage until it reaches the EOF marker, which signifies the current frame (or set of frames) have been encoded and it is ready to output the end sentinels.

In order to compress data into modulo byte-size packages there is a Bit Packer Function embedded in the module which will take the current bits and save them to a register if the data is not byte-size yet. It will append data onto this “bit-overflow” until it reaches this size, and any remainder bits that do not meet the byte-size nature of the output will be added to this register for sequential encoding. Why are we outputting byte-size packages? Well, because of the fact that we may get a sequence of 8 binary ‘1’, in which the decoder might interpret this as a sentinel and ruin the encoding process. The way to get around this, as embedded deep in the ITU T.87, is to add a binary ‘0’ immediately after this set of ‘1’s, which allows the decoder to differentiate between a normal encoded byte and the sentinel. This is a main cause for the data output size being increased to 64-bits.

There is also the case of Limit Overflow Encoding which can happen during run interruption mode or regular mode encoding. The two variants take a different approach but follow the same principle, a series of 0’s will be appended followed by 9 bits of the MErrval – 1 value. The LO Encoding module is essentially just taking the output of the Rice Encoder and orienting bits in the correct order depending on internal flags. Data output may vary in this case with regular limit overflow always being 32-bits, but run interruption mode ranging from 38 to 53 bits (varying J values). Furthermore, the LO encoding for run interruption is further broken up depending on the previous mode. If the previous mode was a non-run mode (mode != 1) then we will have to perform the encoding scheme according to Code Segment A.16 of ITU T.87, along with the LO encoding scheme as described above. In this case however Run Count will always be zero so the only variations that comes is the number of zeros that needs to be encoded first.

Once the output has reached an appropriate size, the FSM will signal a data ready flag, which will be used by an external module to grab the MSB’s of the output depending on the encoded length size.

Implementation

The Bit Packer FSM accomplished the encoding scheme through two internal modules. The Bit Packer Function module performs non LO Encoding, whereas the LO Encoding covers the later case. To ensure there is a modulo byte-size output we maintain an internal register called previous_byte_overflow along with previous_byte_overflow_length, in which bits are appended in descending order from the MSB down. This data is fed through to the internal modules and is appended onto the currently input encoded data depending on the previous_byte_overflow_length.

Bit Packer Function takes the previous byte overflow parameters along with the currently encoded data and length and composes an output set in descending order from the MSB down. The max data output size with appended '0's is 56>, so the data output size is increased to 64 to ensure that any overflow data can be captured by the data output bits. The inner-workings of the module starts with the MSB byte and maps it to the MSB bytes of the data output. Due to the sequence of 8 '1's that was mentioned above we must have an internal check for the upper previous byte to see if it equals 255. If this is the case than a binary '0' is appended to the MSB of the current byte and the encoding continues. This will continue down until the LSB byte. To map the correct overflow byte to the current overflow output we create a sequence of 8 wire busses which will take the different bytes of the final encoded pixel and map it to the overflow data output depending on the data output length. The data output length takes into account the currently input encoded data length, previous encoded data length, and number of appended zeros and sums them up to obtain the final output. This is the length that is used to determine what the final output length along with the correct overflow byte is. All of these sub-mappings are done in parallel, but each mapping must follow a priority encoded scheme due to conditional checks. The data ready flag is used by the Bit Packer FSM to determine when the output is ready to be obtained by the external module.

LO Encoding performs the fairly symmetric encoding of the limited-length Golomb codes. The LO Encoding module actually contains two internal modules, along with a barrel shifter to perform this action. The regular mode encoding is always the same number of binary '0's encoded, along with 8 bits of MErrval – 1. The only variation that comes into play with the regular mode encoding is the length of the previous byte overflow. The internal module Encode Limit takes care of this sequence by performing a comparator check on the length of the previous_byteoverflow_data_length parameter. Depending on the length, we will take a specified number of bits starting from the MSB down of the previous_encoded_data register, and append the upper 23 bits of the encoded_pixel, along with the number of upper bits of the remainder value that makes the output modulo byte-size. The remaining bits of the remainder value are appended to the MSBs of the current_overflow_data, which is saved to the previous_byte_overflow data at the next positive edge of the clock.

In the case of the run interruption mode the number of zeros to be appended may vary according to step 8 of the Run interruption coding sequence specified in ITU T.87. Summarized, if the previous mode was a run mode then we will follow this scheme exactly, where the number of encoded zeros will be determined by (LIMIT – J[Run Index] – 1) – qbpp – 1, which is mapped to 22 – J[Run Index]. In the case that the previous mode was a non-run mode (mode != 1) then the number of appended zeros will increase, as mentioned above, to 23 (J + 1 + (22 – J)). The number of zeros is already predetermined by the Rice Encoder, and is assigned to the encoded_length value. The zeros will be appended to the MSBs followed by the remainder_value. To accomplish this along with the number of binary ‘0’s varying we use a barrel shifter to shift the remainder value by a specified number of bits. The Barrel Shifter takes the previous byte overflow length along with the encoded length values and performs a subtractor operation to determine the shift amount. It will then pass this shift value into the barrel register along with the remainder value pre-shift data, and will output the shifter remainder value. This output will be shifted the correct amount to account for the number of zeros being appended along with the previous byte overflow data appended. The least number of zeros that can be appended is 6, due to run LO encoding, so the highest shift that will be needed is to the [57] bit place, or 49 left shifts. The Barrel Shifter is created to accommodate the edge cases. After calculating the correct shifted data to be encoded, the internal Encode_EOR_Limit module will perform the same action as the regular mode Encode_Limit module. Due to the variants in the design, I have created these modules to work in parallel to reduce latency time between the conditional checks.

Memories

Context Pixel Memories

Either 2 or 3 memories will need to be employed (depending on the final design decision). The memories need to be dual port, 135 width, with each index being 8-bits.

Context Variable Memory

Five memories will need to be employed. 4 of the memories (A, B,C, N) will need to be 365 depth. The depth can be decreased from 366 to 365 due to the fact that run mode does not use these variables. However, if this is done, then the input Q value, Q_Write, to the memory modue will need to be subtracted by one to account for this. The length of each index will differ depending on the context parameter The other memory (Nn) is only used in run interruption modes, Q == 366 and Q == 367, therefore will only need to be a depth of 2 with each index being 7-bits.

A – A needs to be represented with 13 bits to meet the 8192 requirement.

B - B is clamped to a value between -63 to 0. Therefore, each index must be a 7-bit length.

C - C will maintain a value between 127 and -128 which is consistently incremented or decremented based on the current prediction error. This shows that the value C can be represented with 8 bits.

N – The max value is 64 before it is halved by the threshold comparison. Therefore, a 7-bit value is used.

Nn - The maximum counter is 64 just as with N, therefore a 7-bit value can be used.

I have created a basic implementation in the Context_Memory module. This orientation of this module and the ports of the module should be configured exactly this way. The read and write ports are active-low enabled, so an inversion to the input enable signals needs to be made. The write variable is mapped from the write_Context_Memory outputting from the JPEG_LS module.

Testing

Testbenches have been created for each individual module. There are two MATLAB scripts used to generate the correct input and output test values.

JPEG_test_No_Stall.m is used to generate the values for the unit testbenches. To run it you will need to first run the determine the number of rows you want to encode. The image variable will need to be updated to show the amount of rows (image = zeros(644,135) for 644 rows of encoding). In addition, the height variable will need to be updated to reflect this. At the end of the script, there is a sequence of file I/O commands that print the workspace variables to a .mem file. Then, the decimal2hex.py script is run, which will turn these variables from decimal to binary numbers. After running this once all of the testbenches should be ready to be run.

The JPEG_test_No_stall.m file is created to mirror more of the internal workings of the hardware description of the JPEG-LS module. It takes into account the nuances of the timings and stall-less nature of the pipeline, while throwing normal software princples out, so as to create a more realistic implementation. It does not print the correct output due the output not being compressed into byte-size data packets, with a check for 8 binary '1's in a row. However, the correct Rice Encoder outputs are created by the module so as to be able to check the internal Rice Encoder module. Which internal variables are associated with which modules can be determined by looking at the unit testbenches, and the $readmemb associated with the file output of the variable can be mapped.

JPEGLS_codec.m is a software-oriented approach to the design. It performs the encoding to 100% satisfaction with an encoder built in to ensure the design is fully operational. I will allow you all to read through the script to determine its operation, but it is closely mirroring ITU T.87. This script is used to create the correct byte-size encoding outputs. These outputs are stored in a variables called Compress. This Compress variable is mapped to a bitstream in the FinalTBDataAccum.m script. In addition, the FinalTBDataAccum.m script will take the values of the input testbench and map it to the pixel memory one and two variables, so as to simulate the dual port RF.

An overview for the testbenches is given below:

Unit Testbenches

1. Open the JPEG_test_No_stall.m script
2. Update the cwd to the location in where the Verilog files are located
3. Update the image height along with the height variable to set the number of rows needed to be encoded
4. The i and j variables of the for loop will need to be updated, with i representing depth, and j representing rows
5. Run the JPEG_test_No_stall.m script
6. Run the decimal2hex.py script (python3.7 decimal2hex.py)

Integration Testbench

1. Open the JPEGLS_codec.m script
2. Update the cwd to the location in where the Verilog files are located
3. Find the Converted_Data variable and set the the row amount in the zero-matrix generator command
4. The i and j variables of the for loop will need to be updated, with i representing depth, and j representing rows
5. Run the JPEGLS_codec.m script

6. Open the FinalTBDataAccum.m script
7. Looking at the Compress workspace variable, determine the second to last non-zero value index
8. Take the index and set the upper bound of the for loop with the Compressed_Data_Values embedded inside, this will iterate through the Compress variable and turn it into a binary bitstream
9. Update the upper bound of the for loop of the following two for loops based of the equation:

   $$\text{Upper bound} = (\#\text{ of rows} * 135)/2$$

   This will tell the script how many pixels will be in each of the two external pixel memories based off the number of rows encoded

10. Run the integrationmapping.py script (python3.7 integrationmapping.py)

After performing these commands your variables will be written to a file which has been converted from decimal to binary format. Therefore, the Verilog testbenches can read the values line by line via the $readmemb command. Each of the testbenches have a do script associated with them. If wanted, the BitPacker testbench, BytePackerTB, will need to be updated slightly. I was using the integration testbench to debug that module.