System Level Modeling of a H.264 Video Encoder

Che-Wei Chang, Rainer Dömer

Technical Report CECS-11-04
June 2, 2011

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
(949) 824-8919

cheweic,doemer@uci.edu
http://www.cecs.uci.edu

System Level Modeling of a H.264 Video Encoder

Che-Wei Chang, Rainer Dömer

Technical Report CECS-11-04
June 2, 2011

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
(949) 824-8919

cheweic,doemer@uci.edu
http://www.cecs.uci.edu

Abstract

In this report, we demonstrate the results of our system modeling project involving the encoder of the H.264 advanced video coding(AVC) standard. The goal is to create a H.264/AVC video encoder model with SpecC System Level Design Language (SLDL). First, we briefly introduce the essential features of the H.264/AVC video encoder algorithm and the properties of the JM reference C source codes. We then describe the modifications we have performed to make the JM reference C implementation compliant for SpecC SLDL. Our model has been separated into three major behavior blocks, which are Stimulus, Design, and Monitor, to reflect a proper system design testbench. The design-under-test(DUT) component then has been structured into a hierarchy of communicating behaviors. In this work, we have identified several opportunities for parallel execution in the H.264 encoding which we have explicitly specified in the system model. The report concludes with experimental results that validate the correct functionality of our H.264 encoder model by success full simulation.

Contents

List of Figures

List of Tables

List of Listings

System Level Modeling of a H.264 Video Encoder

CW. Chang, R Domer
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
cheweic,doemer@uci.edu
http://www.cecs.uci.edu

Abstract

In this report, we demonstrate the results of our system modeling project involving the encoder of the H.264 advanced video coding(AVC) standard. The goal is to create a H.264/AVC video encoder model with SpecC System Level Design Language (SLDL). First, we briefly introduce the essential features of the H.264/AVC video encoder algorithm and the properties of the JM reference C source codes. We then describe the modifications we have performed to make the JM reference C implementation compliant for SpecC SLDL. Our model has been separated into three major behavior blocks, which are Stimulus, Design, and Monitor, to reflect a proper system design testbench. The design-under-test(DUT) component then has been structured into a hierarchy of communicating behaviors. In this work, we have identified several opportunities for parallel execution in the H.264 encoding which we have explicitly specified in the system model. The report concludes with experimental results that validate the correct functionality of our H.264 encoder model by success full simulation.

1 Introduction

High level languages like C language provide us a fast path to evaluate our algorithm. However, except for fully sequential behavior, it cannot model system architectures such as parallelism and pipeline structure. Hardware description languages like VHDL or Verilog can perfectly describe those structures, but to model the whole system of complex application and refine it with HDL will be very time consuming. To simplify the system design flow and evaluate the software and hardware as a whole at an early design stage, languages such as System Verilog, SystemC or SpecC are developed to fill this gap between high level language and HDL. SpecC [2] is one of the system level description languages (SLDL) which models the algorithm at higher levels of abstraction (than RTL). In this project, we use SpecC to create the system model of a H.264/AVC video encoder.

The most important feature of a SpecC model is the separation of computation and communication. To accelerate the design of a complex System-on-Chip application, reusability of existing design such as hard/soft Intellectual Properties (IPs) is a key feature to solve this problem. SpecC clearly separates the design model of system-on-chip into two independent parts: computation and communication, which are encapsulated into modules/behaviors and channels respectively. Since computation and communication are dealt with independently, the computation behavior blocks can still be reused without or with only minor modification even if the communication in an existing design does not fit in the requirements of the new project. The communication protocol can be easily exchanged by using another channel with compatible interface. In the same manner, we can also modify the computation behavior blocks to fit new requirements without modifying the communication channels.

Another important feature of SpecC is its capability of modeling parallel structures. In a system constructed with multiple processing elements, pipelining or parallel structure are widely used to gain higher resource utility and better performance. High level programming languages like C language generally are executed sequentially and do not support parallel or pipelining structure. In SpecC, programmer can use ’par’ statement in the program to model the parallel structure in the implementation. In this project, these two features are also used to create our system-level model of a H.264/AVC video encoder.

This report is organized as follows: In Section II we will briefly introduce the H.264/AVC video coding standard and some features of the JM reference C implementation. Section III shows the modification we performed on the JM reference C implementation so that it can be compiled by SpecC. In Section IV we will demonstrate our SpecC model of a H.264/AVC video encoder platform. The parallelism we have exploited in our H.264 encoder model will be described in Section V. The simulation result is shown in Section VI.

2 H.264/AVC Video Encoder Algorithm

Before we demonstrate our SpecC model of H.264/Video encoder, in this section we first describe the algorithm [5] and the reference JM C implementation [3].

2.1 Background

H.264/AVC is the video coding standard of the ITU-T Video Coding Experts Group and the ISO/IEC Moving Picture Experts Group. The main goals of H.264/AVC standard is to enhance the video compression performance and provide a ’network-friendly’ video representation for both ’conversational’ applications (video telephony) and ’non-conversational’ (storage, broadcast, or streaming). Compared with existing standards H.264/AVC has achieved a great improvement in rate-distortion efficiency, and it also provides 17 profiles which support different feature sets for various applications [4] .

2.2 Input Video Data Format

The input of a H.264 video encoder and output of a H.264 video decoder are both in YUV format. In a YUV-format file, the information of brightness, called *luma*, and the information of color, called *chroma*, are stored separately. In a YUV-format file, for every frame in the video stream, there will be one array for storing luma information and two arrays for chroma information. Because the human visual system is more sensitive to luma than chroma, sampling formats in which the chroma component has only one half or one fourth of the number of samples than luma component are proposed. The three major sampling formats are listed below:

**4-4-4 sampling* : The number of chroma samples is the same as the number of luma samples in each of the chroma arrays.

**4-2-2 sampling* : The number of chroma samples is the same as the number of luma samples in vertical dimension, but half in the horizontal dimension.

**4-2-0 sampling* : The number of chroma samples is half of the number of luma samples in both vertical and horizontal dimensions

In these three sampling formats, the precisions for each sample are all 8-bits.

2.3 Coding Structure of H.264/AVC Video Encoder

The basic encoding structure of H.264/AVC is shown in Figure 1. In the following sections, we will briefly describe the functions of these blocks.

In H.264/AVC video encoder, each video slice is encoded using intra-frame or inter-frame prediction to remove the spatial and temporal similarity in pictures.

**Intra-Frame Prediction* : The spatial prediction in H.264 is called Intra-Frame prediction. It makes use of the characteristic that in certain areas in a picture, the pixels values are all the same or vary with little difference. In this case, H.264 encoder predicts the pixel values in the current coded macroblock from the pixel values in its neighbor macroblocks to compress the video data. The reference macroblocks are usually the macroblock above or to the left of the current coded macroblock. In H.264, encoder can alternatively select the block-size of 4x4 or 16x16 in the

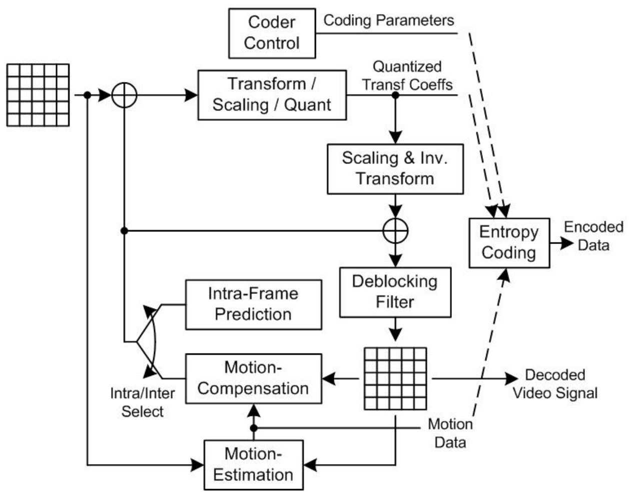


Figure 1: Basic encoding structure for H.264/AVC for a macroblock

```
Mode 0    Vertical Mode
Mode 1    Horizontal Mode
Mode 2    DC Mode
Mode 3    Diagonal Down Left Mode
Mode 4    Diagonal Down Right Mode
Mode 5    Vertical Right Mode
Mode 6    Horizontal Down Mode
Mode 7    Vertical Left Mode
Mode 8    Horizontal Up Mode
```

(a) Prediction Mode of Intra 4x4

```
Mode 0    Vertical Mode
Mode 1    Horizontal Mode
Mode 2    DC Mode
Mode 3    Plane Mode
```

(b) Prediction Mode of Intra 16x16

Listing 1: Modes of Intra-Frame Prediction

inter-frame prediction. For 16x16 and 4x4 block spatial predictions, there are four and nine different ways to predict the pixel value respectively. These prediction modes are listed in Listing 1.

In H.264, the slice which are coded with intra-frame prediction, are called I-slice.

**Inter-Frame Prediction* : The temporal prediction in H.264/AVC video encoder is called Inter-Frame Prediction which makes use of the temporal similarity between current coded pictures and the reference pictures. A continuous video is formed by a sequence of pictures, and a picture is often composed of back scene and the object scene. In a continuous video, the back scene generally is still or varies in very limit difference, and the object scene moves in the continuous pictures in certain regular way (for example in certain speed and direction). Encoder temporal predicts a picture by comparing the difference between current coded picture and reference pictures in the buffer and obtains a motion vector for motion compensation in decoder. In prior standards, the reference pictures are limited to the previous pictures in display order. In H.264/AVC video encoder, this limitation is removed and the reference pictures can be previous or future pictures in display order. The inter-frame prediction performs the block-based motion estimation on every macroblock in the current coded frame, and a distinct motion vector is sent for each macroblock for motion compensation in H.264 decoder.

In H.264, two types of slices could be encoded with inter-frame prediction, which are P-slice and B-slice. In addition to the intra-frame prediction described in the previous section, macroblocks in P-slice and B-slice can be encoded using inter-frame prediction. In inter-frame prediction, both P-slice and B-slice are predicted by motion estimation toward the reference pictures. The major difference between P-slice and B-slice is that unlike P-slice, in which only one motion vector is obtained for motion compensation, in B slice the motion vector is the weighted average of two motion vectors to two different reference pictures.

In H.264/AVC video coding standard, the concept of B slices is generalized. The pixel values are bi-predicted from the weighted combination of two different motion compensated reference frames. Just like the reference pictures in inter-frame prediction, the reference pictures in bi-prediction can be previous or future pictures in display order.

After spatial or temporal prediction, the differences between input pictures and predicted pictures are transformed to frequency domain by the block transformation function in H.264 coding algorithm. There are several unique features about the block transform selected by H.264:

* integer transform design

* specified to 8-bit input video data

* a 4x4 transform size is supported, rather than just 8x8 transform

While the macroblock size is still 16x16, pixels are divided into 4x4 or 8x8 blocks for transformation. The output coefficients are then quantized and scanned (in zig-zag fashion for frame coding) from the lowest frequency coefficient toward the highest. In this way the highest-variance coefficients are ordered first, which maximizes the number of consecutive zero-valued coefficients to gain better performance in entropy coding. In addition to the quantized coefficients, information required to decode the coded data such as encoding parameters, reference frame indexes, and motion vectors for motion compensation are coded with entropy coding and form a compressed video bitstream.

3 Reference C implementation of H.264/AVC Video Encoder

SpecC SLDL is a superset of the C language and it is compliant with ANSI-C. It means that every ANSI-C program can be compiled well by SpecC compiler. To make the H.264 codec modeling easier, rather than developing a H.264/AVC codec with SpecC SDLD from the very beginning, in this project we took the JM 13.0 reference implementation [3], released by Joint Video Team of ISO/IEC MPEG & ITU-T VCEG, as the beginning of this project. Since the JM reference implementation is programmed with C language, the first step to create a SpecC model of H.264/AVC codec in this project, is to modify some syntax in the JM reference implementation which is not ANSI-C or SpecC compliant.

In this section, we briefly describe some features of the JM reference implementation and the modification we did toward the reference implementation for simplicity and SpecC compliant reason.

3.1 Properties of H.264/AVC Video Encoder JM 13.0 Reference Software

The JM reference software of H.264/AVC codec consists of both encoder (in *lencod* directory) and decoder (in *lencod* directory). In this project, we only focus our work on the modeling of H.264 encoder, and use decoder part of the JM reference implementation for consistency verification. Listing 2 shows the properties of the H.264 JM 13.0 reference encoder.

```
Total number of the source files  : 57
Total number of the header files  : 51
Number of the source code lines   : 63K
Number of functions in sourcecode : 814
```

Listing 2: Properties of the H.264 JM Reference Encoder

In H.264/AVC video codec, the video is processed from one picture (frame) to another. Every frame is divided into one or multiple slices, and every slice is then divided into several macroblocks in size of 16x16 luma pixels. Each macroblock is also divided into sub-macroblock partitions for intra and inter frame prediction. For example, the size of sub-macroblock partition for inter-frame prediction can be 16x16, 16x8, 8x16, 8x8, 8x4, 4x8, and 4x4. The transform for the difference between original frame and predicted frame is either in size of 8x8 or 4x4. Figure 2 shows the hierarchy of a video sequence in H.264/AVC video codec.

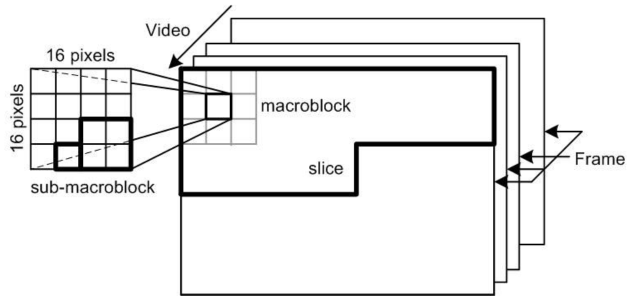
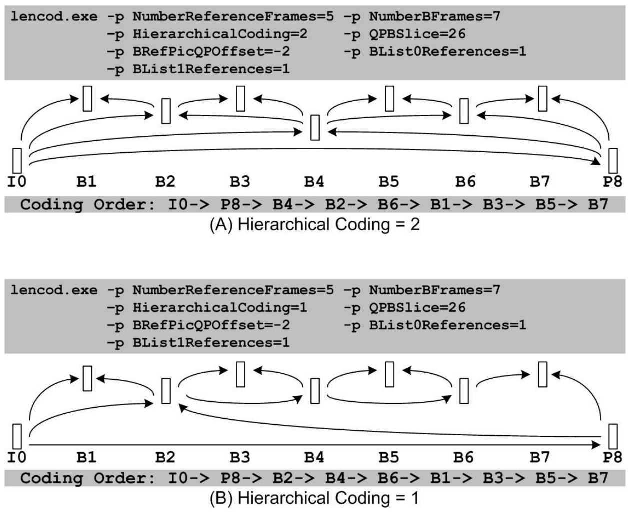


Figure 2: Hierarchy of Video Stream

In the reference C implementation, we can also see the same hierarchy in the source code. A simplified flow chart of the program execution is shown in Figure 3. Frames in the video stream are read out from the YUV file one by one by a *ReadOneFrame* function call and then every slice in the frame is encoded by a *encode_one_slice* function call in a while loop. The slice is further divided into 16x16-pixel macroblocks and every macroblock is then encoded by a *encode_one_macroblock* function call. The coding tools such as inter or intra prediction and spatial transform are all executed in the *encode_one_macroblock* function.

Notes that in this JM reference implementation, the picture frames do not have to be encoded in the display order. According to the JM Reference Software Manual, user can decide the order by modifying the HierarchicalCoding setting in the encoding parameter. Two examples of changing coding order are shown in Figure 4. In the first example, the coding order for nine consecutive frames is frame 0, 8, 4, 2, 6, 1, 3, 5, and frame 7; in the second example, the coding order for nine consecutive frames is frame 0, 8, 2, 4, 6, 1, 3, 5, and frame 7.



```
lencod.exe -p NumberReferenceFrames=5 -p NumberBFrames=7
           -p HierarchicalCoding=2    -p QPBSlice=26
           -p BRefPicQPOffset=-2      -p BList0References=1
           -p BList1References=1
```

```
Coding Order: I0-> P8-> B4-> B2-> B6-> B1-> B3-> B5-> B7
```

```
lencod.exe -p NumberReferenceFrames=5 -p NumberBFrames=7
           -p HierarchicalCoding=1    -p QPBSlice=26
           -p BRefPicQPOffset=-2      -p BList0References=1
           -p BList1References=1
```

```
Coding Order: I0-> P8-> B2-> B4-> B6-> B1-> B3-> B5-> B7
```

Figure 4: Example of Coding Order

3.2 SpecC compliant reference implementation

In this project, we took JM13 version of H.264 video encoder as reference C implementation, and converted it to a hierarchical SpecC model so that we can conduct further design space exploration using SCE [1]. However, the JM reference implementation is not fully ANSI-C compliant and some syntax in the reference C codes will lead to errors when the reference source codes are compiled with SpecC. The first step we have to accomplish is to make some modification to the reference C implementation so that it can be compiled by SpecC compiler. Besides, some function calls, such as *fopen* or *fwrite* are ANSI-C compliant, though, they are not adequate to be used in the SpecC model because they are not hardware-synthesizable.

In this section, we list the modifications we performed on the reference C implementation for creating our SpecC model of H.264 encoder.

1. Encoding Parameters Initialization: In the original reference C implementation, major encoding parameters such frame rate, frame mode selection, and search algorithm selection for motion estimation, are initialized at the beginning of video encoding. During initialization, a configuration file called *encoder.cfg* is opened by a function call *fopen* and a structure named *InputParameters*, which stores the encoding parameters is initialized with the content in *encoder.cfg*. Since *fopen* is not hardware-synthesizable, we either move it out of the H.264 SpecC model or just hard code the parameters in the H.264 SpecC model. To simplify the communication between the *stimulus* behavior (in charge of providing test patterns to the *design* behavior) and *design* behavior (in charge of implementing the major function the target algorithm) here we simply hard-coded the parameters in the *design* behavior. More details about these two behavior and the communication between them will be described in the next section.

The same situation also happened in quantization table initialization. Instead of reading the quantization coefficients from configuration file *q_it* matrix.cfg and *q_offset.cfg*, we hard-coded the coefficients into the tables in *design* behavior.

2. Variables Initialization: This part can be separated into local variables initialization and global variables initialization. Unlike C program, in which variables can be initialized to the result of certain arithmetic or logical operations, at declaration the variables can only be initialized to a constant values in SpecC SLDL. In local variable initialization, we can fix this difference by simply manually separating the variable declaration and initialization. For global variables, since assigning values to variables outside of function scope is not SpecC compliant, we created two functions called *sei_init* and *post2ctxinit* to fix this problem. At the beginning of encoding procedure, these two functions will be called to initialize global variables. Listing 3 shows in general how we modified the variable initialization.

3. Function Pointer Elimination: In C language,

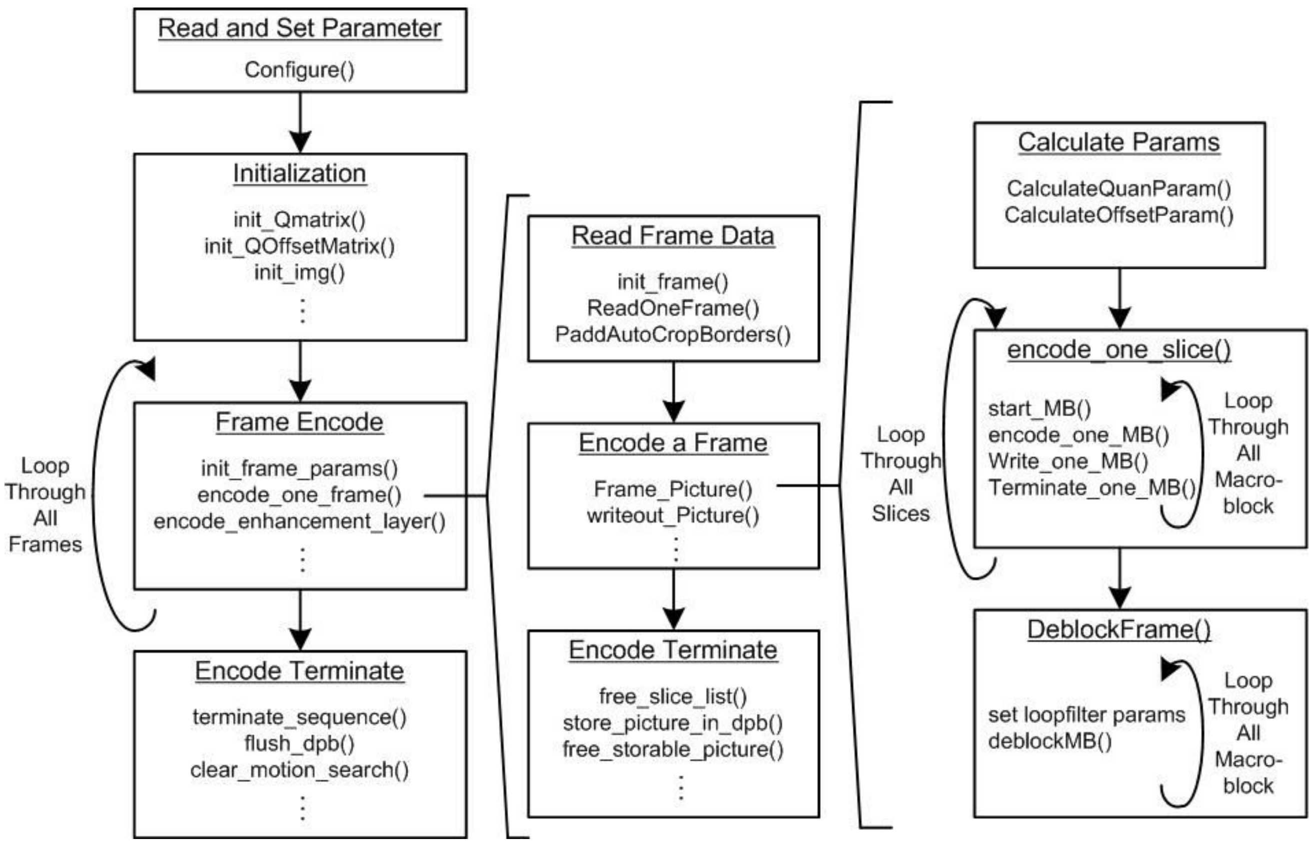


Figure 3: Code Structure

function pointer is very useful for program simplicity and maintenance, which can greatly reduce the use of if or case-switch statements. However, due to its ambiguity for high-level synthesis, function pointers are not adequate in SpecC. In the JM H.264 reference software, the function pointer syntax is widely used due to various encoding options implemented in the JM reference implementation. To eliminate the function pointers in the program, we either removed the encoding options or replaced them with a switch statement (even though it makes the program look more complex). When function pointers are used for encoding option selection, for simplicity, we only kept one encoding option and eliminated the others. Listing 5 and Listing 6 shows how we replaced the function pointers with case-switch statement. Listing 4 shows the existing function pointers in the JM reference implementation.

4. Source files renaming: Since SpecC compiler only accepts source files with *.sc* postfix, 53 source files are renamed with same name and postfix by *.sc*.

After these modifications, the reference implementation is SpecC compliant. Since the encoding parameters are hard coded in the program already, except for YUV file, no other input file is needed.

4 Modeling of H.264/AVC Video Encoder Platform

One important feature of SpecC is the separation of computation and communication, which increases the reusability of existing software and hardware implementation. In this section, we also describe our SpecC model of H.264 video encoder in these two views.

First, we will describe the functions of three major behaviors at the topmost level in every SpecC model: *stimulus*, *design*, *monitor*. Then, in the second sub-

```
var_type global_var1 = global_var1_init ;
var_type global_var2 = global_var2_init ;
                    :
                    :

return_var_type function_name(input_vars)
{
  var_type local_var1 = local_var1_init;
  var_type local_var2 = local_var2_init;
                    :
                    :

 (funciton description)
}
```

(a) Before Variables Initialization Modification

```
var_type global_var1 ;
var_type global_var2 ;
          :
          :

void global_var_init()
{
  global_var1 = global_var1_init ;
  global_var2 = global_var1_init ;
                :
                :
}

return var_type main (input_vars)
{
  local variable declaratoin
           :
  global_var_init() ;
           :
 (function description)
}

return_var_type function_name(input_vars)
{
  var_type local_var1 ;
  var_type local_var2 ;
                 :
  local_var1 = local_var1_init ;
  local_var2 = local_var2_init ;
                 :
 (funciton description)
}
```

(b) After Variables Initialization Modification

Listing 3: Variables Initialization Modification

```
void (*writeMB_typeInfo)
void (*writeIntraPredMode)
void (*writeB8_typeInfo)
void (*writeRefFrame[6])
void (*writeMVD)
void (*writeCBP)
void (*writeDquant)
void (*writeCIPredMode)
void (*writeFieldModeInfo)
void (*writeMB_transform_size)
int (*WriteNALU)
int64 (*getDistortion)
extern imgpel *(*get_line[2])
extern imgpel *(*get_line1[2])
extern imgpel *(*get_line2[2])
extern imgpel *(*get_crline[2])
extern imgpel *(*get_crline1[2])
extern imgpel *(*get_crline2[2])
extern int (*computeUniPred[6])
extern int (*computeBiPred)
extern int (*computeBiPred1[3])
extern int (*computeBiPred2[3])
void (*getNeighbour)
void (*get_mb_block_pos)
```

Listing 4: List of function pointers in JM reference encoder

section, we will describe the communication between *stimulus* & *design* and *design* & *monitor*.

4.1 Major Behaviors : Stimulus, Design, Monitor

In HDL implementation, to verify the correctness of an implementation, a testbench environment including the function of providing test pattern and function of output comparison is necessary. This concept is also applicable to SpecC modeling. At the topmost level of a SpecC model, two behaviors named *stimulus* and *monitor* are created to implement the function of the testbench, and the application we want to model with SpecC is encapsulated in a behavior named *design*. These three behaviors usually are executed in parallel, and the communication between them are implemented to guarantee the necessary synchronization among these three behaviors.

In the following paragraphs, we describe the functions of these three behaviors and the roles they play in our SpecC model of H.264/AVC video encoder.

```
return_type *fp(input vars) ;

return_type func_1(input vars)
{
          :
}

return_type func_2(input vars)
{
          :
}

void fp_init (init_condition)
{
          :
  switch(condition){
    condition_1:
      fp = func_1 ; break ;
    condition_2:
      fp = func_2 ; break ;
    default:
  }
}

return_type fp_call_1(input vars)
{
           :
         fp() ;
           :
}

return_type fp_call_2(input vars)
{
           :
         fp() ;
           :
}
```

Listing 5: Before Function Pointer Elimination

```
return_type func_1(input vars)
{
          :
}

return_type func_2(input vars)
{
          :
}

return_type fp_call_1(input vars)
{
          :
  switch(condition){
    condition_1:
      func_1(input vars); break ;
    condition_2:
      func_2(input vars); break ;
    default:
  }
          :
}

return_type fp_call_2(input vars)
{
          :
  switch(condition){
    condition_1:
      func_1(input vars); break ;
    condition_2:
      func_2(input vars); break ;
    default:
  }
          :
}
```

Listing 6: After Function Pointer Elimination

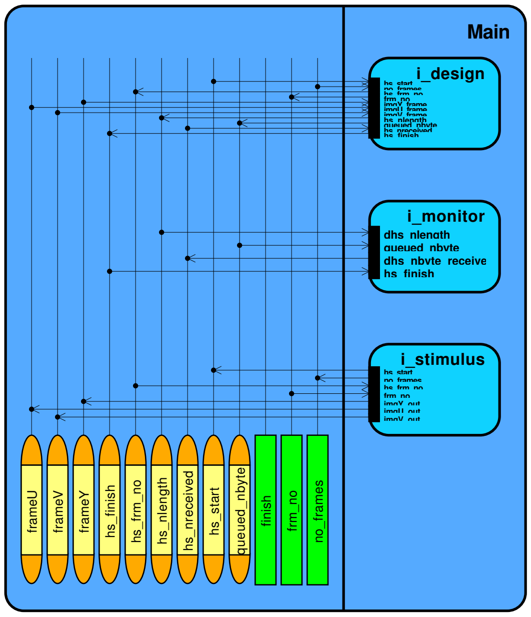


Figure 5: Top level of H.264 Encoder Model

* Behavior *Stimulus* :

In a SpecC model, this behavior is in charge of providing input data for *Design* behavior. It can be considered as the test pattern generator for design verification. Depending on the implementation of *Design* behavior, *Stimulus* behavior sometimes is also in charge of providing configuration information for *Design* behavior. Since the encoding parameters and the quantization coefficients for H.264 video encoder are hard-coded in our *Design* behavior, *Stimulus* behavior does not have to read parameters and coefficients from the configuration file.

In the original JM reference implementation, a variable named *FrameNumberInFile*, which indicates the next frame to be encoded, will be updated for each frame encoding and a function named *ReadOneFrame* is called.

This function will read the corresponding picture frame from a opened video raw file named *test.yuv*. Note that in the JM reference implementation the frame coding order does not have to be the same as the display order. Therefore, *FrameNumberInFile* variable indicating the next picture frame to be encoded, is necessary. In our model, the *Stimulus* behavior is implemented based on the *ReadOneFrame* function. At the beginning of the encoding process in our SpecC model, *Stimulus* behavior will open raw file *test.yuv* for read with function *fopen*. After the YUV file is opened, *Stimulus* behavior will notify *Design* behavior of the beginning of video encoding. *Stimulus* behavior then enters a state of waiting for *FrameNumberInfile* updating from *Design* behavior. Once *Stimulus* behavior receives *FrameNumberInfile*, it reads the corresponding picture frame from file *test.yuv* and sends frame data to *Design* behavior over the communication channels. In this project, for simplicity, we have fixed the sampling format to YUV420 and the frame size at 176x144 for Y frame and 88x72 for U and V frames.

* Behavior *Design under test (DUT)* :

In our SpecC model, this behavior is in charge of the implementation of target design, and it is also the behavior in which we will perform the design space exploration in the future. *Design* behavior receives input data from *Stimulus* behavior, performs the application or algorithm on the received data and then generates output results for validation. The content of *Design* behavior depends on our design goal: we can implement a specific part of the target application or algorithm, such as temporal/spatial prediction or spatial transform, and only evaluate the execution of this specific part; we can also implement the whole target application or algorithm in *Design* behavior and evaluate the execution of the whole algorithm.

In our project, *Design* behavior implements the whole H.264/AVC video encoding algorithm. For every frame, *Design* behavior generates a new *FrameNumberInfile* and send it to *Stimulus* behavior to request a new set of frame data, and then enters a state of waiting for corresponding frame data. Once corresponding Y, U, and V frame data are all received, *Design* behavior performs the spatial/temporal prediction, spatial transform and quantization, and entropy coding over the received frame data. The encoded data is then sent to *Monitor* behavior over communication channels in the form of byte-sequence for generating a H.264-format file named *test.264*. The transmission of the encoding data is performed on frame basis, i.e., for every frame, *Design* sends a sequence of bytes to *Monitor* behavior. Note that because of the different characteristics between picture frames, the length of the byte-sequence for encoded data varies. Since *Monitor* behavior is independent of the execution of H.264 video encoding, it does not have the information about how long the byte sequence is. Therefore, except for the encoded data, *Design* behavior also has to send the corresponding length of byte sequence to *Monitor* behavior for every encoded frame. Once every picture frame in the input raw file is encoded, *Design* issues a notification to *Monitor* behavior so that *Monitor* behavior knows when to terminate the encoding.

In our *Design* behavior there are two sub-behaviors named *h264_encoder* and *h264_writer* working in parallel to achieve the task described above: *h264_encoder* is the behavior in charge of the implementation of H.264 video encoding algorithm, and *h264_writer* deals with the transmission of length of byte-sequence to *Monitor* behavior. When the encoding of a frame is finished, *h264_encoder* will send the byte-sequence to *Monitor* through a byte-queue and the corresponding length of byte-sequence is sent to *h264_writer* through a integer-sequence. Once every frame in the YUV file is encoded, *h264_encoder* will send out a length of byte-sequence with zero value to *h264_writer*. The function of *h264_writer* is to read the length of byte-sequence from the integer-queue and send the value to *Monitor*. *h264_writer* also detects the zero value of the length. When a length of byte-sequence with zero value is detected, *h264_writer* will notify the monitor of the end of encoding. Figure 6 shows the block diagram of *Design under Test* generated by System-on-Chip Environment (SCE).

* Behavior *Monitor* :

The main function of this module is to generate output file(s) for verification. Depending on the implementation of *Design* behavior, the content of the output files varies. If *Design* behavior implements one specific part of the target implementation or algorithm, then the content of the output files should

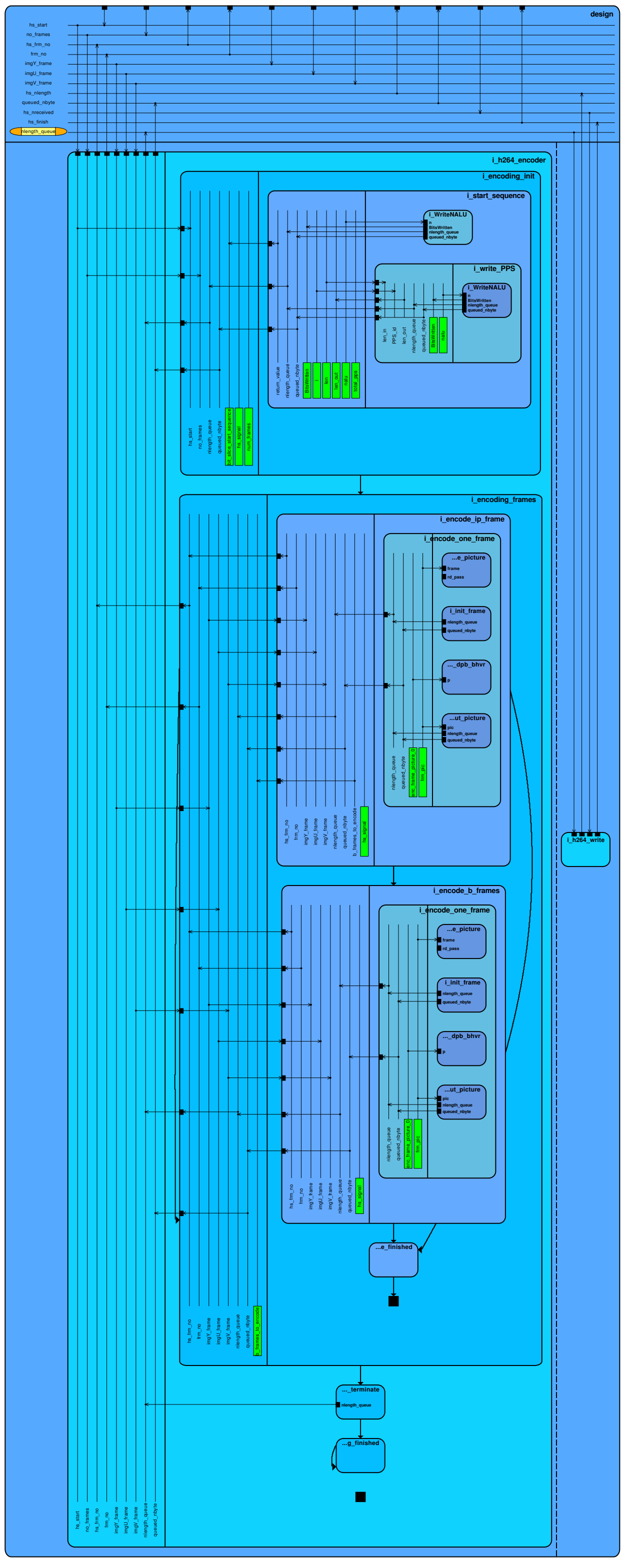


Figure 6: Design under Test Behavior

be the output of this specific part; if *Design* behavior implements the whole application or algorithm, the content of the output file is the output of the application or algorithm. According to the implementation of *Design* behavior, golden sample(s) are pre-generated and compared with the output file(s) for consistency. In this project *Design* behavior implement the whole H.264/AVC video encoding algorithm, which makes the generation of gold sample(s) easier: we simply use the encoded video stream generated by JM reference software as golden sample.

In original reference C implementation of H.264 encoder, when a frame encoding is finished, a function called *WriteNALU* will be called. In this function the encoded data bytes will be *fwrite* to a output file *test.264*. In SpecC model, *fwrite* operation is moved out from *Design* to *Monitor* behavior because *fwrite* function is not hardware-synthesizable. The main function of *Monitor* behavior in our SpecC model is to receive byte-sequence from *Design* behavior and write those bytes to output file *test.264*. Another function implemented in *Monitor* is to terminate the encoding program with *exit(0);* which is not hardware-synthesizable either.

To satisfy the requirements described above, there are two separate sub-behaviors named *monitor_write* and *monitor_finish* which work in parallel in *Monitor* behavior. The *monitor_write* behavior is in charge of the byte-sequence reading and writing the bytes to the output file *test.264*; the *monitor_finish* behavior is in charge of the termination of the whole application. After the initialization of the H.264 video encoding in our SpecC model, these two behaviors are both in waiting state and wait for the length of byte-sequence and end of encoding notification, respectively. *monitor_write* behavior has to receive the length of next encoded byte-sequence from *Design* behavior through a double_handshake channel, before it fetches corresponding number of bytes from a byte queue between *Design* and *Monitor* behavior and *fwrite* the bytes into *test.264*. During the video encoding *monitor_write* is always in waiting state. When it receives the end of encoding notification from *Design* behavior, it closes the output file and uses *exit(0)* to terminate the program.

4.2 Communication between top-level behaviors

* *communication between Stimulus and Design* :

The communication between stimulus and design modules is implemented with two double handshake channels and three frame queues. The first handshake channel is used to send encoding initiation signal from *Stimulus* to *Design* to start the encoding; the second double handshake channel is used to transfer variable *FrameNumberInFile* from *Design* to *Stimulus* to request frame data. Once *Stimulus* receives the *FrameNumberInFile* variable from *Design*, it will read the corresponding frame and send them to *Design* through frame queue channels. The Y, U, and V frames are transmitted separately through different frame queue channel. For simplicity, the size of the frame in our encoder is fixed to 176x144 pixels per frame. Therefore, the size of frame queue channel is 176x144 bytes for Y-frame queue, 88x72 bytes for U and V frame. For now, in frame encoding, the depth of these three queues are all set to 1.

Figure 7 shows the comparison between JM reference software and SpecC model, and the communication between *Stimulus* and *Design* behaviors as well.

**communication between Design and Monitor* :

The communication between design and monitor is implemented with one handshake channel, two double_handshake channels, and one byte-queue channel. The two double_handshake channels are used to synchronize the update of length of byte-sequence in *h264_writer* and bytes writing operation in *monitor_writer*. To update the length of byte-sequence *h264_writer* in it Design behavior will read one integer from the integer queue between *h264_encoder* and *h264_writer*. If the length of byte-sequence is zero, *h264_writer* will notify the *Monitor* of the end of encoding through the handshake channel, and *Monitor* will terminate the encoding program immediately; if the length of byte-sequence is not zero, *h264_writer* will send the value to *Monitor* through a double_handshake channel HS_NLENGTH. After the transmission, *h264_writer* will enter a waiting state and wait for the reply from *Monitor* through another double_handshake channel named HS_NRECEIVED.

When *Monitor* finishes the writing operation, it sends an acknowledge through HS_NRECEIVED to notify *h264_writer* of the accomplishment of writing operation. After receiving the acknowledge, *h264_writer* continues the update of length of byte-sequence.

Figure 8 shows the comparison between JM reference software and SpecC model, and the communication between *Design* and *Monitor* behaviors as well.

5 Exploiting parallelism in H.264 Video Encoder

H.264 video codec is a very complex algorithm with heavy a computational load. For some applications, such as video camera which has to encode input video in real-time, we might have to add parallel structure in the implementation to accelerate the encoding. After inspecting the reference JM source code of H.264 encoder, we have found some possible parallelism in the encoder.

In this section, we have two parallel structures implemented in our H.264 encoder model.

One important feature of SpecC SLDL is its ability to simulate parallel structure. Parallel structure is very common in hardware description language like Verilog, but for high level languages like C, it is quite difficult to simulate parallel structure since programs are usually executed in sequential way. In SpecC, *par* statement is used when concurrent execution is required.

One simple instance of concurrent execution with *par* statement is shown in Listing 7. With this feature, designer can explore parallelism in the target application and verify the functionality without actually implementing it with hardware description language.

```
behavior B_par
{
  B  b1,  b2,  b3;

  void main(void)
  {
        par{
                b1.main();
                b2.main();
                b3.main();
        }
  }
};
```

Listing 7: Concurrent Execution with par statement

Dependency between codes implies sequential execution order, and in most cases disturbing this order leads to functionality error. Therefore, we have to make sure that there is no dependency between the variables used in the code before executing two or more sections of code, which was running sequentially originally, in concurrent way. Due to the prediction mechanism in the H.264 video encoder, parallelism across frames and macroblocks might not be feasible: intra-prediction of a macroblock requires the pixel values from the left, up, and up-left macroblocks, and inter-prediction of a macroblock uses the macroblocks in the previous reconstructed frames as references to compute the motion vectors.

Two possible opportunities for parallelism we have found are concurrent encoding of multiple slices, and parallel execution of encoding procedure inside one macroblock. In this report, for now we only focus on the parallelism inside macroblock encoding. The two occurrences of parallelism, which we have implemented in our H.264 encoder model, are (1) Luminance/Chrominance residual coding and reconstruction, and (2) Motion vector search for multiple reference frames.

5.1 Luminance/Chrominance residual coding and reconstruction

The first potential for parallelism we found in the H.264 encoder is the residual coding of Luma(Y) and Chrominance(U/V) pixels in a macroblock. In intra- and inter-prediction encoding, the difference between predicted macroblock and original macroblock, called residual, will be transformed with DCT and then quantized to reduce the size of data. Then the transformed and quantized residual will be written into output file. Since the quantization in H.264 codec is lossy quantization, to make sure the reference frames in the encoder are identical to the reference frame in the decoder, the quantized residual in the encoder has to be inverse-quantized and inverse-transformed so that the residual in the encoder and decoder end are identical. The predicted macroblock then will be combined with inverse-quantized and transformed residual to reconstruct the decoded picture. The decoded picture will be put in decoded picture buffer for future reference in inter-prediction. During the residual coding and reconstruction, luminance pixels and chrominance pixels are processed separately. In original *RDCost_for_macroblocks* function, the luma pixel coding and chroma pixel coding are executed sequentially. In our H.264 SpecC model, we managed to combine the luma and chroma residual coding and reconstruction in inter-prediction coding and execute it concurrently.

The code before and after modification are shown in Listing 8. In the original source code, we can see that function *ChromaResidualCoding* will be called after luma residual coding except for *IPCM* mode. Now we have created a new behavior *LumaChromaResidualCoding_bhvr* to concurrently execute luma and chroma residual coding in inter-prediction encoding. In the future, we will add luma residual coding in intra-prediction encoding(mode==I16MB and I8MB) into the concurrent execution as well. Another potential for parallelism in this part is that the U and V chroma pixel residual coding can be further separated and executed in parallel.

5.2 Motion vector searching for multiple reference frames

In H.264 encoder, the reference pictures are stored in two decoded picture buffers named *List0* and *List1*. The predicted macroblock in inter-prediction is obtained by comparing the current macroblock with the macroblock in the reference frame and searching for the reference macroblock with minimal error. Inter-prediction will search reference pictures in decoded picture buffer and find out the motion vector between original macroblock and best-matching reference picture. For P frame encoding, all reference pictures in *List0* will be searched for inter prediction; for B frame encoding, both decoded picture buffer *List0* and *List1* will be searched.

The parallelism we exploited in inter prediction is

```
func  RDCost_for_Macroblocks (..)
{
                    :
  if  (mode<P8x8)
    LumaResidualCoding(currMB);
  else  if  (mode==P8x8)
    SetCoeffAndReconstruction8x8(currMB);
  else  if  (mode==I16MB)
    Intra16x16_Mode_Decision(currMB);
  else  if  (mode==I8MB)
    Intra8x8Macroblock(currMB);
  else  if  (mode==IPCM)
                    :

  if(mode!=IPCM)
    ChromaResidualCoding(currMB);


                    :
}
```

(a) Original sequential Luma/Chroma Residual Coding

```
behavior  LumaChromaResidualCoding_bhvr(..)
{
  par{
    i_LumaResidualCoding_bhvr.main();
    i_ChromaResidualCoding_bhvr.main();
  }
};

behavior  RDCost_for_Macroblock_bhvr(..)
{
                    :
  if  (mode<P8x8)
    i_LumaChromaResidualCoding_bhvr.main();
  else  if  (mode==P8x8)
    SetCoeffAndReconstruction8x8(currMB);
  else  if  (mode==I16MB)
    Intra16x16_Mode_Decision(currMB);
  else  if  (mode==I8MB)
    Intra8x8Macroblock(currMB);
  else  if  (mode==IPCM)
                    :

  if((mode!=IPCM)&(mode>=P8x8))
    i_ChromaResidualCoding_bhvr.main();


                    :
};
```

(b) Luma/Chroma Residual Coding in Parallel

Listing 8: Residual Coding Parallelization

in the function called *PartitionMotionSearch*. This function in the original C code calculates the minimal error and finds out the best motion vector for each reference picture. In our H.264 encoder model, we create a parallel behavior to execute the motion vector search for all reference pictures in the decoded picture buffer. Since the number of reference pictures is user-defined and we set this number to five, there will be at most five parallel motion vector searches for P frame encoding and ten parallel motion vector searches for B frame encoding in our H.264 model.

The codes before and after modification are shown in Listing 9. Note that the behavior *RefFrameMotionSearch_bhvr* in parallel model is not identical to the behavior in the sequential model. In parallel version of motion vector search, every *RefFrameMotionSearch_bhvr* will be assigned a list number and reference picture number. Before executing the motion search, these two numbers will be compared with variables *num_list* and *num_ref* to determine if the motion vector search in the corresponding reference picture and list is required. We did this modification to prevent motion vector search from being executed when the reference picture does not exist. For example, if there are only two reference pictures in the decoded picture buffer and the current frame is a P frame, only *RefFrameMotionSearch_bhvr_00* and only *RefFrameMotionSearch_bhvr_01* will execute the motion vector search, and the rest *RefFrameMotionSearch_bhvr* will be idle.

```
behavior PartitionMotionSearch_bhvr(..)
{
  for (list=0; list < num_lists; list++)
  {
    for (ref=0; ref < num_ref; ref++)
    {
                  :
        i_RefFrameMotionSearch_bhvr.main() ;
                  :
    }
  }
                  :
} ;
```

(a) Sequential Motion Vector Searching

```
behavior Parallel_RefFrameMotionSearch_bhvr(..)
{
                 :
  par{
    i_RefFrameMotionSearch_bhvr_00.main() ;
    i_RefFrameMotionSearch_bhvr_01.main() ;
    i_RefFrameMotionSearch_bhvr_02.main() ;
    i_RefFrameMotionSearch_bhvr_03.main() ;
    i_RefFrameMotionSearch_bhvr_04.main() ;
    i_RefFrameMotionSearch_bhvr_10.main() ;
    i_RefFrameMotionSearch_bhvr_11.main() ;
    i_RefFrameMotionSearch_bhvr_12.main() ;
    i_RefFrameMotionSearch_bhvr_13.main() ;
    i_RefFrameMotionSearch_bhvr_14.main() ;
  }
} ;

behavior PartitionMotionSearch_bhvr(..)
{
                 :
  i_Parallel_RefFrameMotionSearch_bhvr.main();
                 :
} ;
```

(b) Parallel Motion Vector Searching

Listing 9: Motion Vector Searching Parallelization

6 Experiments and Results

In this project, we validate our H.264 SpecC model by comparing the output video file generated by our model with the output file generated by reference C code. We also calculate the run time of the encoding process of our H.264 model. A simulation report of the H.264 video encoding for a 19-frame video clip with our H.264 SpecC model is shown in Figure 9.

In the simulation report, some parameters of this encoder model are shown: the input video format is YUV420, the frame size is 176x144, the number of reference frames is 5, and so on. We can also see that the encoding order is different from displaying order in this simulation report. The encoding order in this simulation is I0, P2, B1, P4, B3, P6, B5, P8, B7, P10, B9, P12, B11, P14, B13, B16, B15, P18, and then B17. The run time of this simulation is also shown after the encoding is finished.

H.264 video encoder might be the most complicated application we have ever modeled with SpecC. In this simulation, there are 19 frames been encoded: one for I-frame and nine for P-frame and B-frame individually. Since the frame rate of this video clip is 30 frames per second, the total length of the encoded video is about 2/3 second. To encode such a short video clip, it took about 90 seconds to complete the process with our H.264 model.

7 Conclusion and Future Work

In this project, we have implemented a system model of a H.264/AVC video encoder with SpecC SLDL. We first made necessary modifications in the JM reference software to make the reference source code SpecC compliant. After the reference code is fully SpecC compliant, we separated the reference implementation into three major behaviors *Stimulus*, *Design*, *Monitor* in our SpecC model, and assigned proper channels for the communication between these three behaviors.

We have also exploited two opportunities for parallelism in our H.264 encoder model: parallel luminance and chrominance pixel residual coding, and parallel motion vector search for multiple reference pictures.

At this point, our H.264 SpecC model is still far away from a perfect synthesizable SpecC model. There is still lots of work to do to make it perfect. Several improvements we plan for the future are listed below:

1. Global variables elimination: Global variables are very convenient for C language programmer and also makes C programs look concise. Every function in the C program can access the global variables without explicit variable transmission. However, it is not appropriate to use global variables in SpecC because of the lack of explicit communication. To let System-on-Chip Environment (SCE) estimate the communication cost in design exploration, we should keep every variable transmission as explicit as possible. Therefore, the global variables have to be replaced by local variables and corresponding communications in the SpecC model. Right now the communication between the three major behaviors is totally global-variables free, but we still have to eliminate the global variables in *Design* behavior in the future.

2. Memory allocation elimination : In an embedded system where memory space is fixed and limited, memory allocation functions such as malloc are not hardware-synthesizable. Therefore, it is better to replace memory allocation functions with explicit array declarations in the SpecC model.

3. Explore more parallelism in *Design* behavior: We have found two opportunities for parallelism and implemented them in our H.264 model. There might be more parallelism to be exploited in the encoder. For example, since both intra-frame and inter-frame prediction are used in P-slice encoding, we could look into this part to find out if it is possible to run these two predictions in parallel. Another potential parallelism we can explore is to add pipeline structure in our SpecC model. Since the encoding of a macroblock is processed in the order of temporal/spatial prediction, transform and quantization, and then entropy coding, maybe it is possible to execute these steps in pipeline structure.

4. Design space exploration on System-on-Chip Environment (SCE) : System-on-Chip Environment (SCE) is a system developing tool in which we can assign different combinations of processor(s) and hardware(s) to implement the H.264 encoder and estimate the execution time for those different resource assignments and operating frequencies. After we create a ”clean” SpecC model in which all global variables are eliminated and all functions are hardware-synthesizable, we plan to explore the design space of H.264 video encoder with SCE for different requirements and applications.

Acknowledgment

This work has been supported in part by funding from the National Science Foundation (NSF) under research grant NSF Award #0747523. The authors thank the NSF for the valuable support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] L. Cai, A. Gerstlauer, S. Abdi, J. Peng, D. Shin, H. Yu, R. Dömer, and D. Gajski. System-onchip environment (sce version 2.2.0 beta): Manual. Technical Report CECS-TR-03-45, Center for Embedded Computer Systems, December 2003.

[2] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[3] H.264/AVC JM Reference Software. http://iphome.hhi.de/suehring/tml/.

[4] Wikipedia H.264/MPEG-4 AVC. http://en.wikipedia.org/wiki/H.264/MPEG-4_AVC.

[5] Gisle Bjontegaard Thomas Wiegand, Gary J. Sullivan and Ajay Luthra. Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7), JULY 2003.

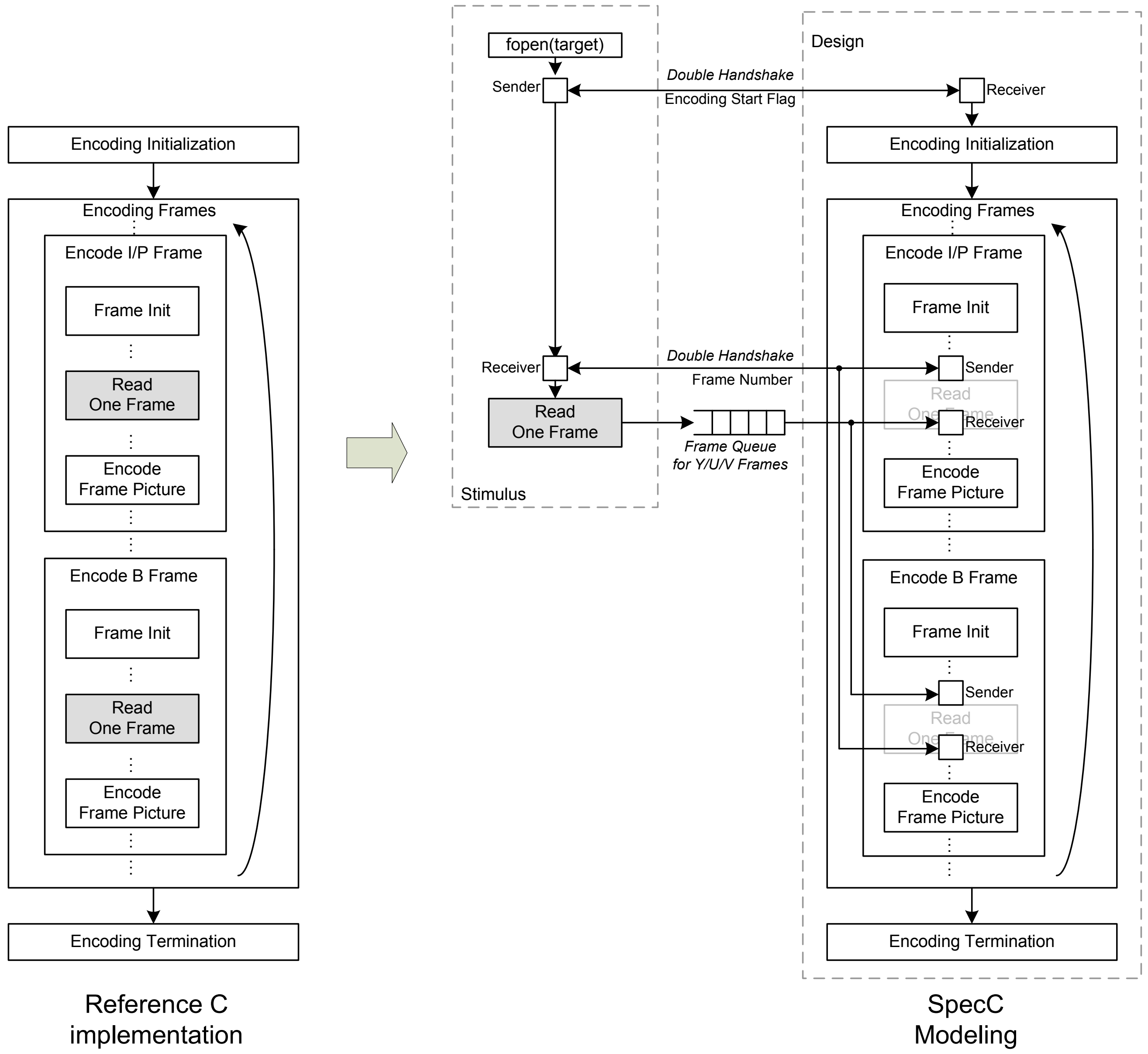


Figure 7: Communication between Stimulus and Design

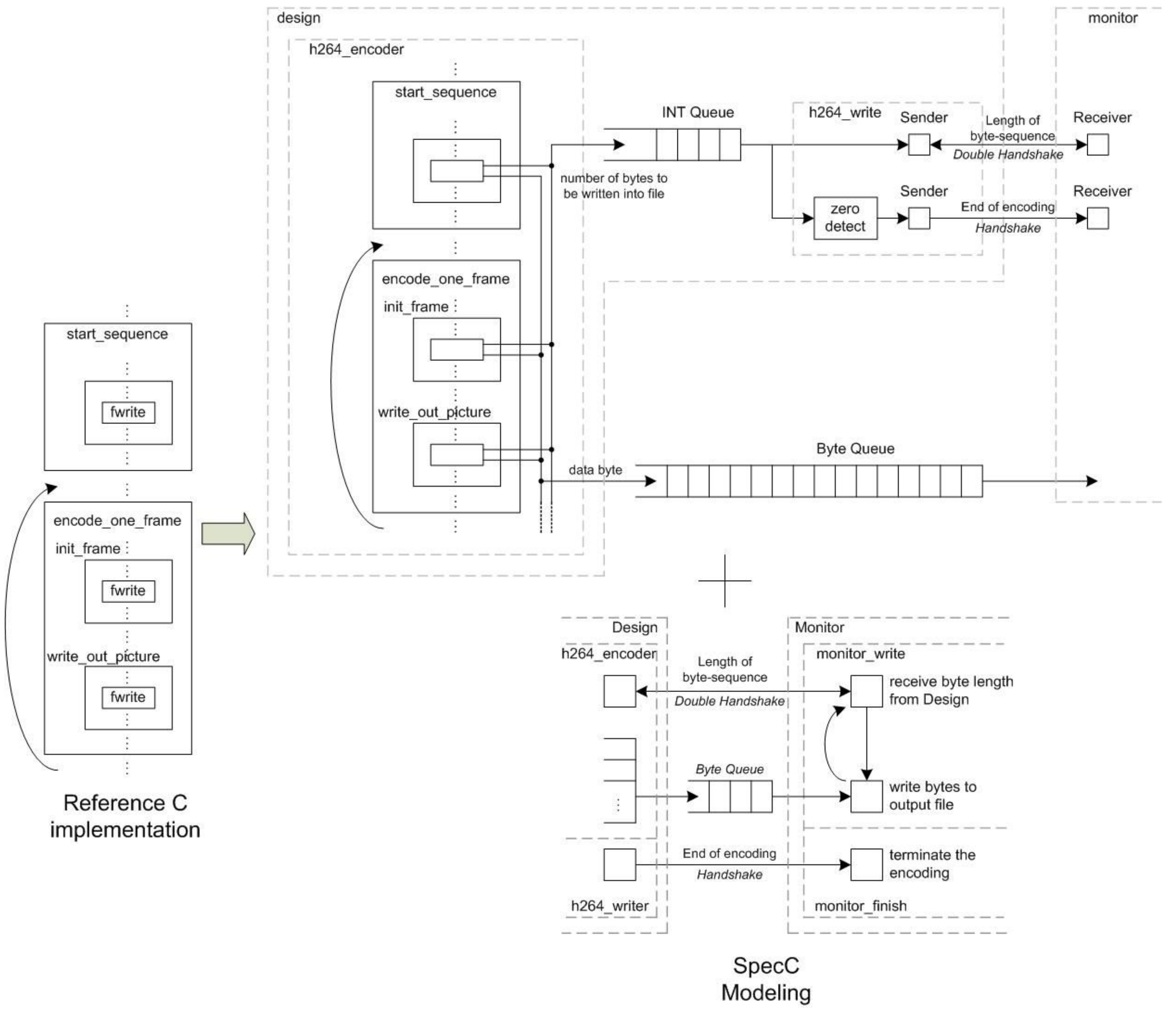


Figure 8: Communication between Design and Monitor

```
------------------------------------- H.264 SpecC Model -------------------------------------
 Input YUV file                          : test.yuv
 Output H.264 bitstream                  : test.264
 Output YUV file                         : test_rec.yuv
 YUV Format                              : YUV 4:2:0
 Frames to be encoded I-P/B              : 10/9
 Freq. for encoded bitstream             : 15
 PicInterlace / MbInterlace              : 0/0
 Transform8x8Mode                        : 1
 ME Metric for Refinement Level 0        : SAD
 ME Metric for Refinement Level 1        : Hadamard SAD
 ME Metric for Refinement Level 2        : Hadamard SAD
 Mode Decision Metric                    : Hadamard SAD
 Motion Estimation for components        : Y
 Image format                            : 176x144
 Error robustness                        : Off
 Search range                            : 32
 Total number of references              : 5
 References for P slices                 : 5
 List0 references for B slices           : 5
 List1 references for B slices           : 5
 Sequence type                           : I-B-P-B-P (QP: I 28, P 28, B 30)
 Entropy coding method                   : CABAC
 Profile/Level IDC                       : (100,40)
 Motion Estimation Scheme                : Fast Full Search
 Search range restrictions               : none
 RD-optimized mode decision              : used
 Data Partitioning Mode                  : 1 partition
 Output File Format                      : H.264 Bit Stream File Format
-------------------------------------------------------------------------------------------------

Encoding. Please Wait.

=============== FrameNumberInFile =  0, Type = I_SLICE ===============
=============== FrameNumberInFile =  2, Type = P_SLICE ===============
=============== FrameNumberInFile =  1, Type = B_SLICE ===============
=============== FrameNumberInFile =  4, Type = P_SLICE ===============
=============== FrameNumberInFile =  3, Type = B_SLICE ===============
=============== FrameNumberInFile =  6, Type = P_SLICE ===============
=============== FrameNumberInFile =  5, Type = B_SLICE ===============
=============== FrameNumberInFile =  8, Type = P_SLICE ===============
=============== FrameNumberInFile =  7, Type = B_SLICE ===============
=============== FrameNumberInFile = 10, Type = P_SLICE ===============
=============== FrameNumberInFile =  9, Type = B_SLICE ===============
=============== FrameNumberInFile = 12, Type = P_SLICE ===============
=============== FrameNumberInFile = 11, Type = B_SLICE ===============
=============== FrameNumberInFile = 14, Type = P_SLICE ===============
=============== FrameNumberInFile = 13, Type = B_SLICE ===============
=============== FrameNumberInFile = 16, Type = P_SLICE ===============
=============== FrameNumberInFile = 15, Type = B_SLICE ===============
=============== FrameNumberInFile = 18, Type = P_SLICE ===============
=============== FrameNumberInFile = 17, Type = B_SLICE ===============
H264 Encoding Process has finished
Total Run Time: 90.000 seconds
Files test.264 and test.264.gold are identical
```

Figure 9: Simulation report