Process Network Modeling and TLM Generation for H.264 Codec Design

Yongjin Ahn, Samar Abdi

Technical Report CECS-08-11
Oct. 10, 2008

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-4922

http://www.cecs.uci.edu

Process Network Modeling and TLM Generation for H.264 Codec Design

Yongjin Ahn, Samar Abdi

Technical Report CECS-08-11
Oct. 10, 2008

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-4922

http://www.cecs.uci.edu

Abstract

Process network model has been widely used for system specification because of its modeling capability. It allows relatively easy modeling of a system, especially when the system design is initially captured by a set of computation blocks connected by the flow of data, which is common in multimedia applications. It also allows efficient representation of concurrency of a system, and is often used as a model for mapping an application to architecture with multiple processing elements.

In this report, we present how to create the process network model in SystemC from the given H.264 application in C. We first present H.264 algorithm in brief and show how we can model the process network for H.264. We have manually implemented three

process network models in SystemC for H.264, that is, H.264 encoder, H.264 decoder, and H.264 codec. Then, they have been tested in ESE (Embedded System Environment) FrontEnd tool which has been developed for efficient design of multiprocessor architectures on system-on-chips. We have created several platforms to test them in ESE FrontEnd, generated TLMs (Transaction Level Models) and simulated them successfully. The main purpose of this report is to explain how to model process networks from the scratch and help someone who wants to start with ESE FrontEnd from process network model.

Table of Contents

List of Figures

List of Tables

Chapter 1 :Introduction

Process network model originated from [1] has been widely used since it allows relatively easy modeling of a system and efficient representation of concurrency of a system, and is often used as a model for mapping an application to a multiprocessor architecture. In the process network model, all processes communicate only with FIFO (First In First Out) buffers which are theoretically unbounded. Because of the unbounded FIFOs, there are some issues on executing the process network model such as the buffer size and deadlock. However, in this report, we assume that the deadlock problem can be solved by designer and the buffer size is given even though they are not optimal. (Refer to [2] for details).

We explain how we can describe the process network model in SystemC [3] from the given application in C. The process network model can also be implemented in C but in this report, we use SystemC because it provides the FIFO channels so that it is easy to create and verify the process network model. This report is organized as follows.

Chapter 2 presents how we can describe the process network model in SystemC using a simple example. Chapter 3 explains what H.264 is and what features have been newly improved compared to the previous standards. Chapter 4 and 5 show the process network models in SystemC for H.264 encoder and H.264 decoder respectively. Then, chapter 6 explains how we can create H.264 codec from them. Chapter 7 presents how to create platforms and generate TLMs from the process network models. Chapter 8 shows the simulation time and code size of all the process network models and the generated TLMs by ESE FrontEnd. Finally, we conclude this report in chapter 9.

Chapter 2 :Process Network Model in SystemC

In order to model a process network from a given application, designers need to analyze the application and divide the application into several processes which may have the possibility of concurrency. Once the application partitioning is done, we can create the process network model in SystemC.

2.1 Modeling Processes

First of all, since we are focusing on the functional model in SystemC, we should use only untimed SystemC features which do not include timed constructs such as *clk*, *wait*, *sensitives*, and so on.

Second, all processes must communicate each other only using SystemC FIFOs. (*sc_fifo* or *sc_port* not *sc_in* or *sc_buffer*) It means that two processes cannot share any global variable and they must read or write the value of the variable only via a FIFO channel. If a process comes from a function which has several arguments, then all the arguments must be transformed to FIFO channels.

Third, SystemC is just used for wrapping processes. Therefore, it does not have to include all codes from C. If the process calls some functions, the functions can be described in other C files.

We use an example to explain in more detail how to create each process in SystemC as shown in Figure 1. Let us assume that an application is given as the upper figure part in Figure 1. And assume that from analyzing the main function in ‘main.c’, we can divide

the application into 3 processes. In that case, we can create each process in SystemC as shown in the bottom figure part in Figure 1. For ‘Process A’, we need to add one FIFO channel because ‘Func_A’ and ‘Func_B’ share a global variable. And for ‘Process B’, we need two FIFO channels, one for a global variable from ‘Process A’ and the other for an argument toward ‘Process C’. In case of ‘Process B’, we have integrated the codes in ‘funcB.c’ into the SystemC code. However, like ‘Process C’, other functions can still exist in other C files.

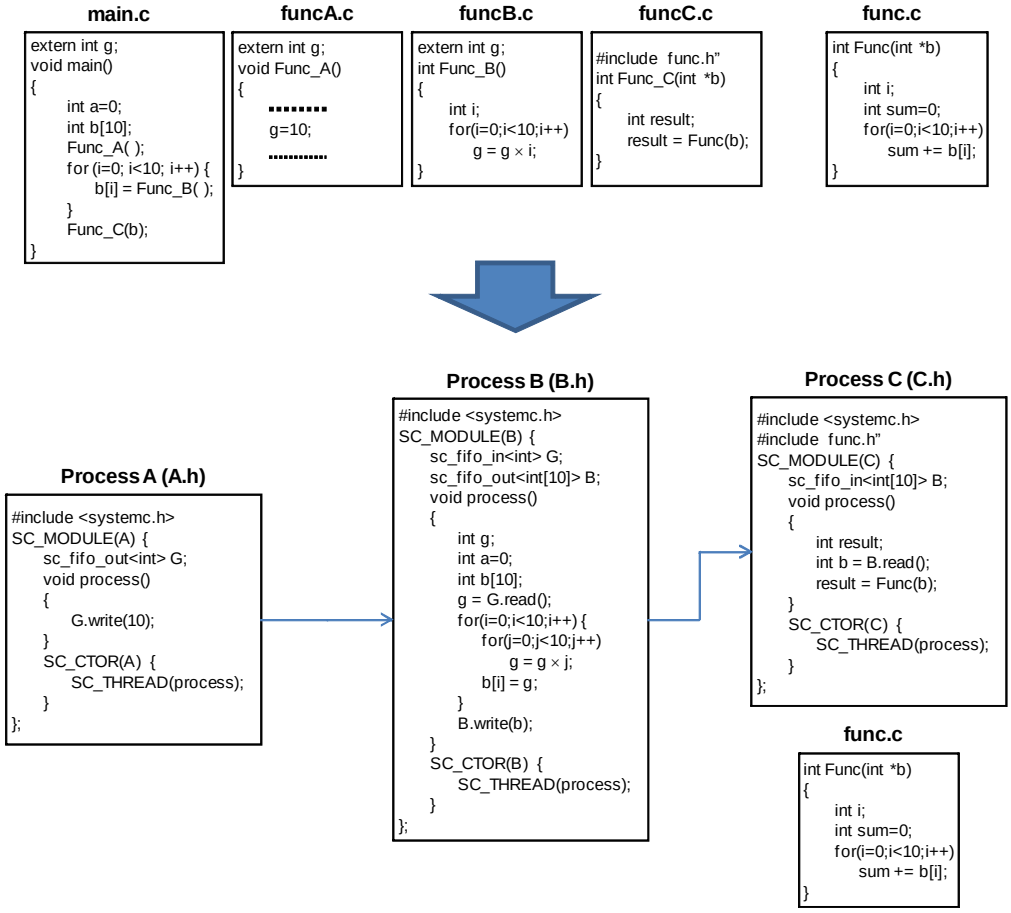


Fig 1. The SystemC processes created from a simple application.

2.2 Top SystemC Code

Figure 2 shows the top SystemC code which describes the whole system of the application in Figure 1. First, three processes (A, B, and C) are instantiated and then two FIFO channels (R1 and R2) are created. Finally, the two FIFO channels connect three processes. For example, a FIFO channel ‘R1’ connects a port ‘G’ of process ‘a’ (a→G) and a port ‘G’ of process ‘b’ (b→G).

```
#include <systemc.h>
#include "A.h"
#include "B.h"
#include “C.h”
int sc_main(int argc, char* argv[])
{
        A *a;
        B *b;
        C *c;
        a = new A(“a”);
        b = new B(“b”);
        c = new C(“c”);

        sc_fifo <int> R1(“R1”,1);
        sc_fifo <int[10]> R2(“R2”,1);
        a->G(R1);
        b->G(R1);
        b->B(R2);
        c->B(R2);

        sc_start(-1);
        return 0;
};
```

Fig 2. The top SystemC code.

Chapter 3 :H.264 Algorithm

Now, we need to analyze H.264 application in order to implement the process network model of it. We start with the H.264 reference code written in C which is available at http://www.itu.int. The profile used in our H.264 software is the baseline profile which is used widely in videoconferencing and mobile applications. The details in algorithms can be found in [4]. Figure 3 shows the block diagram of H.264 encoder and Figure 4 shows the block diagram of H.264 decoder. H.264 consists of four main parts, that is, motion estimation/compensation (ME/MC), transformation (DCT, Quantization, etc.), deblocking filter, and entropy coding.

First, in the motion estimation/compensation (ME/MC) part, the current frame is described based on the previous frame. The estimated frames are called “InterFrames” and some frames are encoded without any prediction, which are called “IntraFrames”. H.264 has more features than H.263 which is the previous standard. In case of the ME/MC, it considers variable block sizes to predict current frame and can take multiple and arbitrary reference frames.

Second is the transformation and quantization part. H.264 uses integer linear transformation which is faster and more accurate than discrete cosine transformation used in the previous standard.

Third, as a deblocking filter, H.264 uses the adaptive in-loop deblocking filter to reduce the blocking phenomenon.

Finally, entropy coding is based on the statistical estimation which finds out an optimal codeword from the frequency of a symbol. In the entropy coding part, H.264

adapts a new algorithm called context adaptive variable length codes.

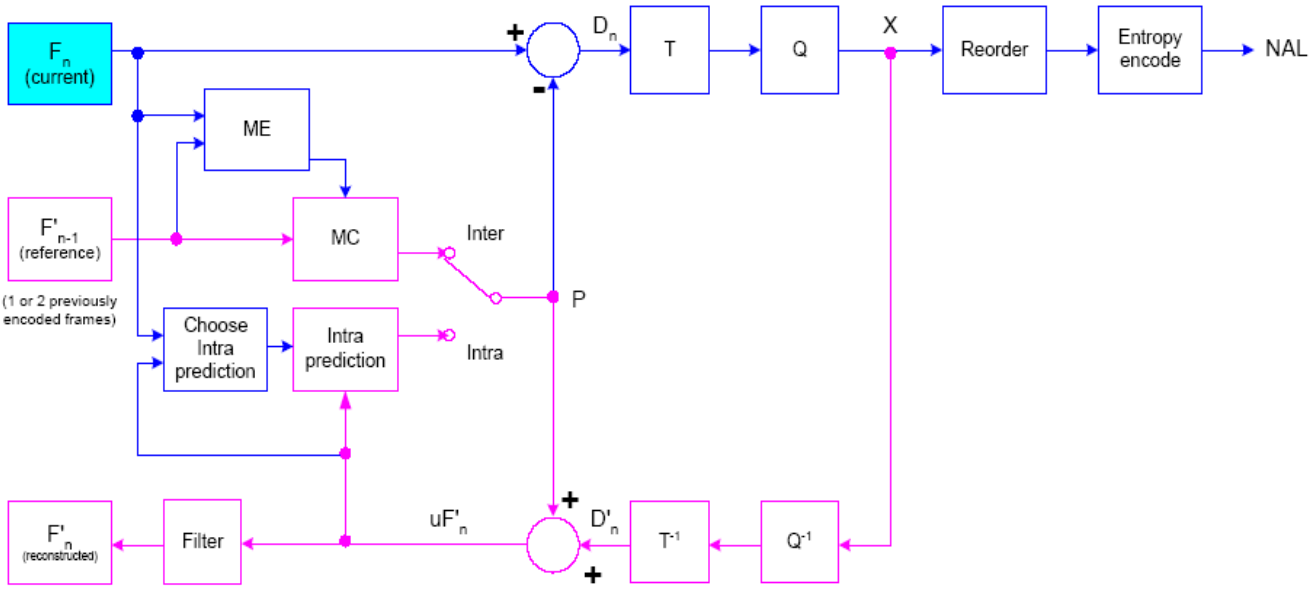


Fig 3. The block diagram of H.264 encoder.

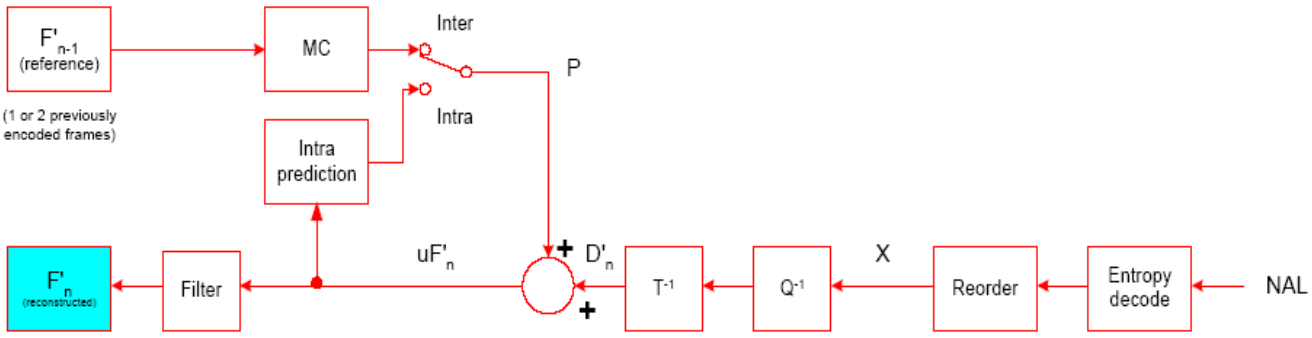


Fig 4. The block diagram of H.264 decoder.

Chapter 4 :Process Network Model for H.264 Encoder

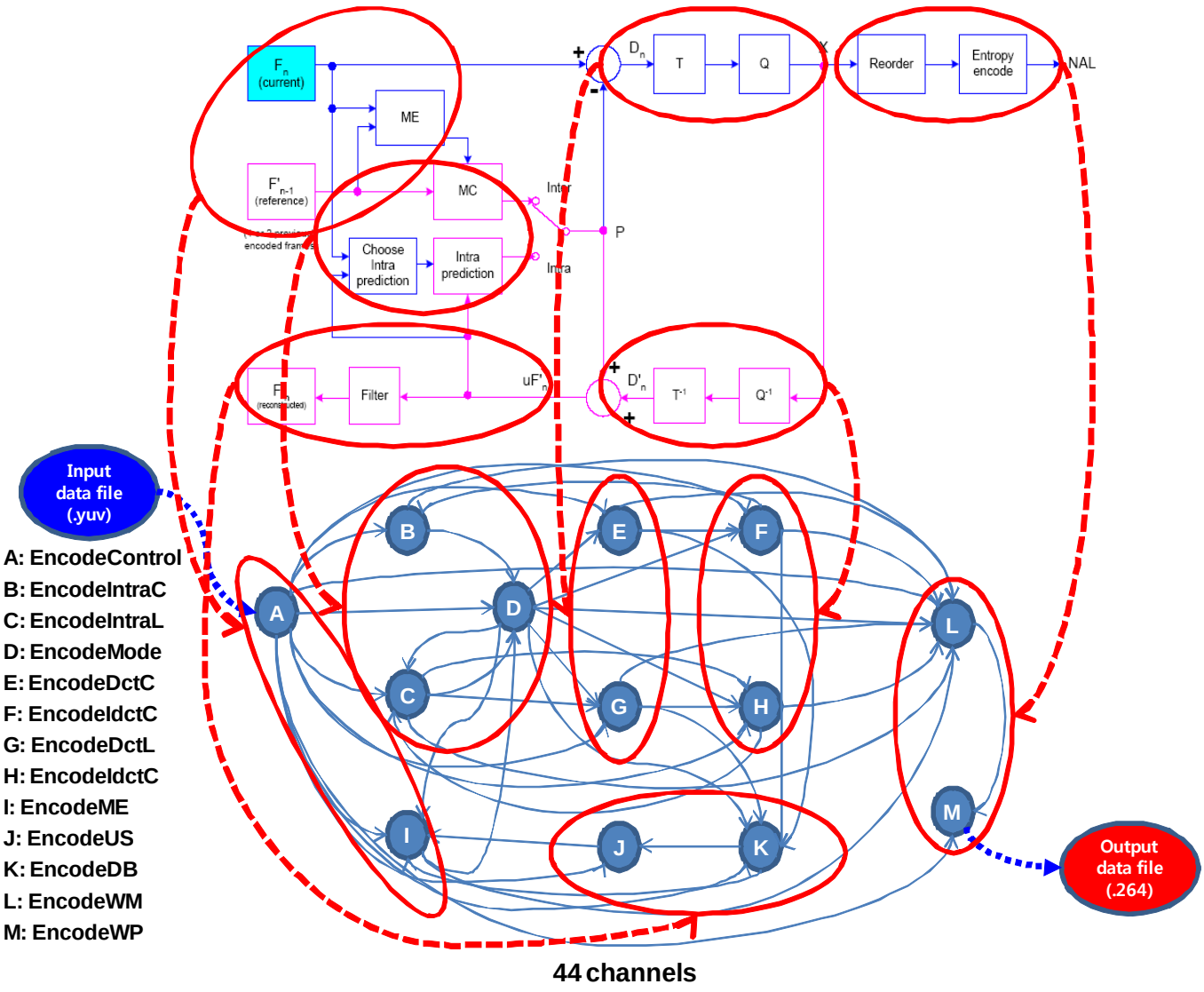


Fig 5. The process network model graph of H.264 encoder.

Based on the block diagram shown in Figure 3, we have divided the reference code into many processes in order to create the process network model in SystemC. Figure 5 shows the process network model graph of H.264 encoder. It also shows the relationship between the process network model and the block diagram in Figure 3. There are 13 processes and 44 channels. All processes have been created in SystemC in the same way as shown in Figure 1. The input of H.264 encoder is a YUV file (*.yuv) which is a raw format and its output is a 264 file (*.264) which contains encoded data.

Chapter 5 :Process Network Model for H.264 Decoder

In the similar way to H.264 encoder, we have created the SystemC process network model for H.264 decoder. Figure 6 shows the process network model graph for H.264 decoder and the relationship between the process network model and the block diagram in Figure 4. There are 12 processes and 23 channels. The input is a 264 file and the output file is a YUV file.

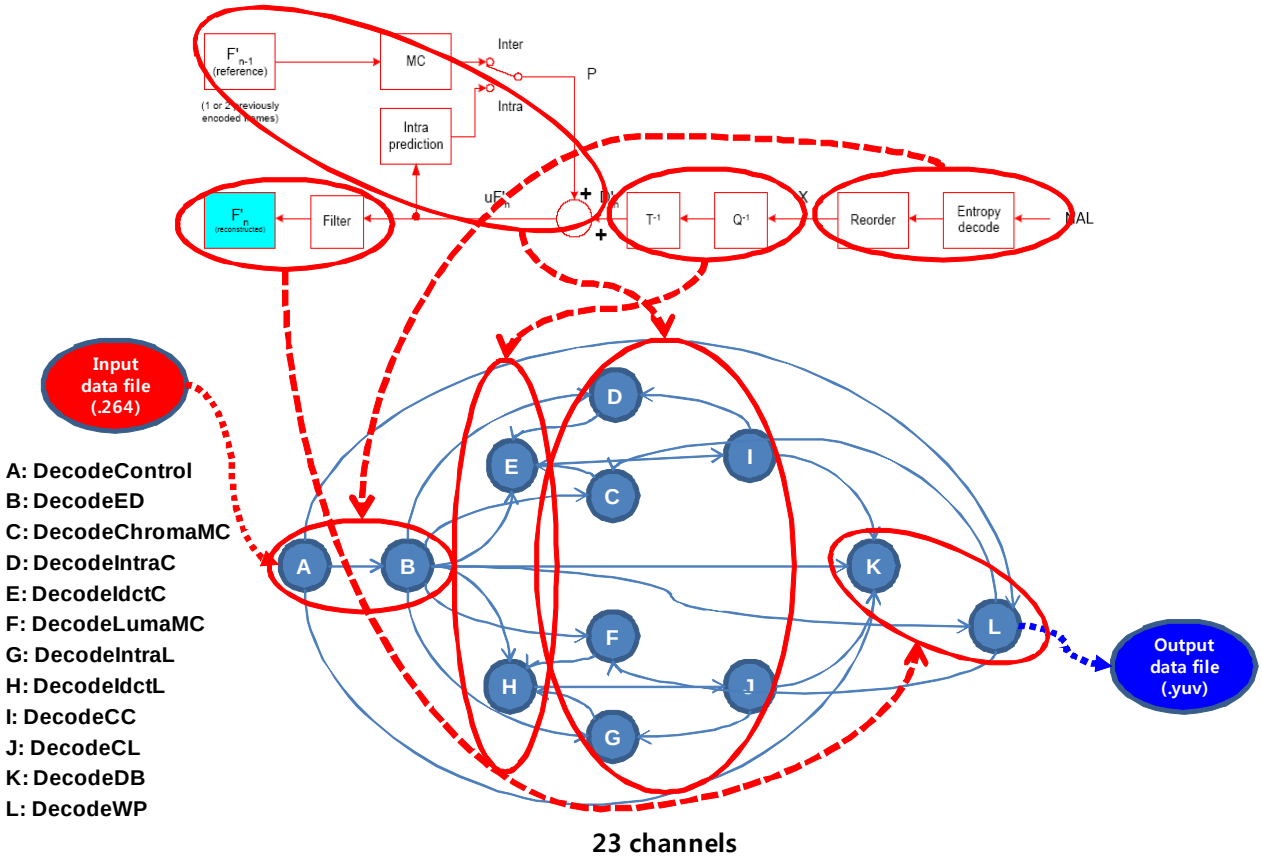


Fig 6. The process network model graph of H.264 decoder.

Chapter 6 :Process Network Model for H.264 Codec

In this chapter, we explain how to create the process network model for H.264 codec. Since we have both H.264 encoder and decoder, only what we need to do is merging them into one process network model.

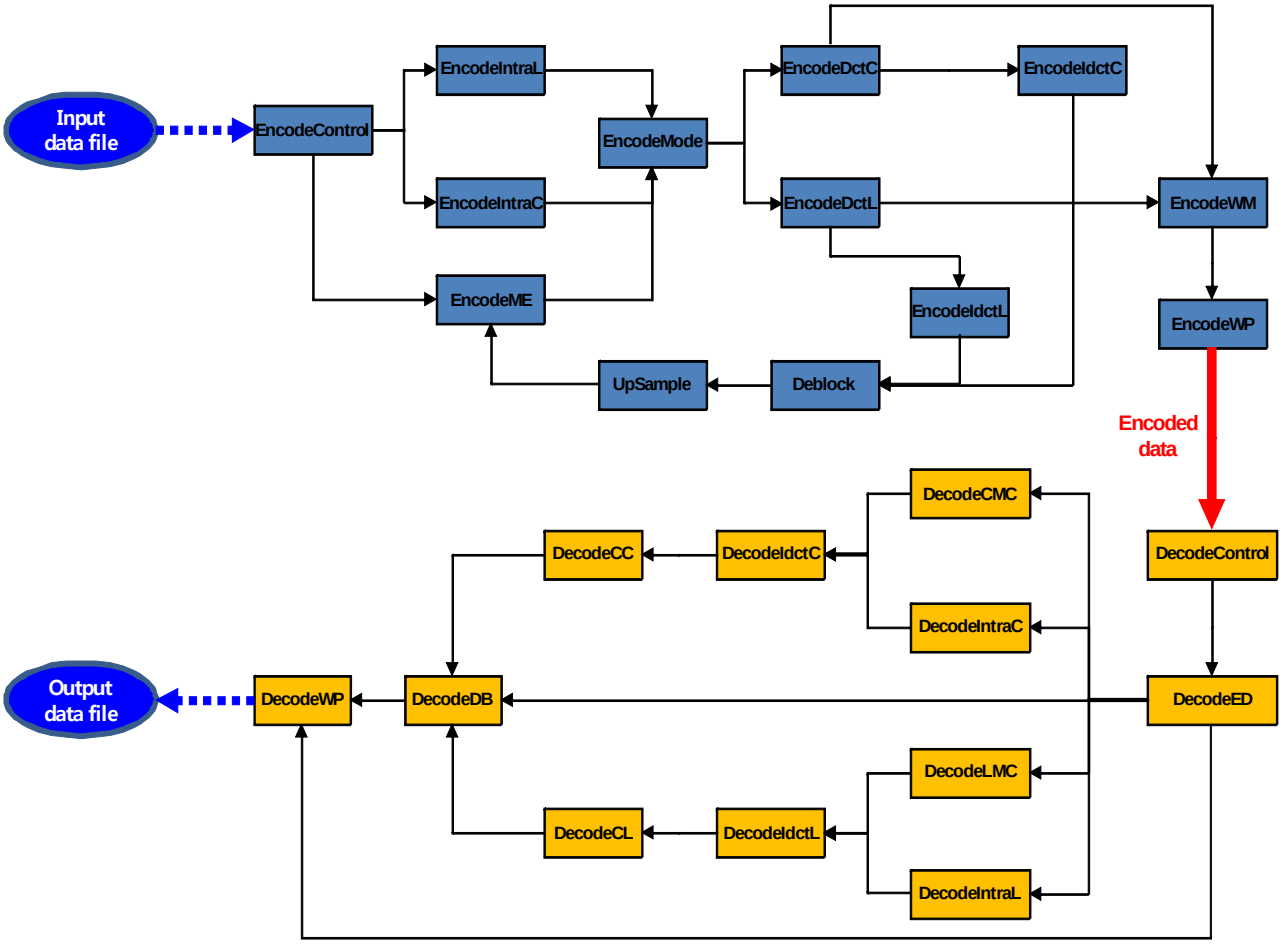


Fig 7. The process network model of H.264 codec.

Since H.264 encoder outputs an encoded file and H.264 decoder inputs the encoded file, we need to remove those file interfaces, and we can add FIFO channels to connect them properly. H.264 encoder writes the encoded data frame by frame to a file in the ‘EncodeWP’ process and similarly, H.264 decoder reads the data frame by frame from a file in the ‘DecodeControl’ process. Therefore, we can insert a FIFO buffer whose size is the maximum among the frames in order to connect encoder and decoder.

Based on this analysis, we have removed all file interfaces in the two processes, added a FIFO channel to them and merged them together. The final process network model for H.264 codec is shown in Figure 7. (Also see Appendix A)

Chapter 7 :Platform Creation and TLM Generation in ESE FrontEnd

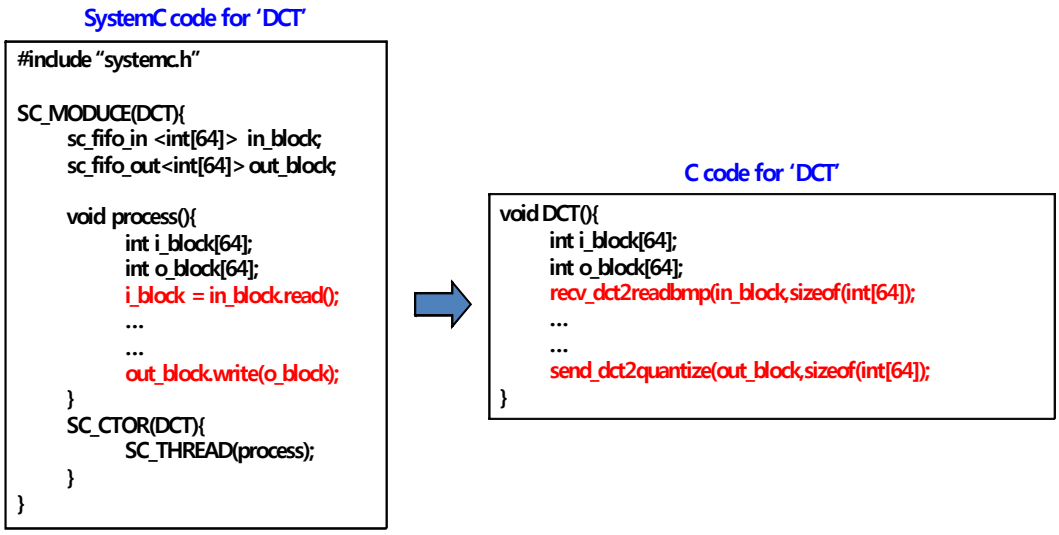
In this chapter, we present how to create platforms in ESE FrontEnd and how to generate the TLMs. First of all, in order to input the process network model to ESE FrontEnd, we need to extract the input C code from the SystemC code. Note that we can directly create the process network model from the reference C code. If we already have C code for each process of the process network model, we do not have to create the input C code from the SystemC code. Or, we can directly create and verify the process network model by using ESE FrontEnd.

With the input C code, we can create various platforms in ESE FrontEnd. Once the platforms are created by designers, ESE FrontEnd can generate two types of TLMs in SystemC that is, the functional TLM and the timed TLM. For the timed TLM, ESE FrontEnd estimates the execution time of each process in the process network model and annotates the timing delay into the original code [2]. And then, the timed TLM is evaluated by simulation, thus the overall performance of the system is reported to the designer.

7.1 C Code Creation for ESE FrontEnd

As mentioned above, in order to input a process network model to ESE FrontEnd, we go through a process to generate C code for each process. We explain it using an example process named ‘DCT’ (in JPEG encoder) in SystemC as shown in Figure 8 (a). It has two FIFO channels, one for receiving data and the other for sending data. From the SystemC

code, we remove all SystemC dependant statements and exchange the FIFO read/write functions to the functions defined in ESE FrontEnd as shown in Figure 8 (b). We need specify the name of signal and the size of it. In this manner, we can generate all C codes for all processes.



(a) (b)

Fig 8. (a) SystemC code for 'DCT' (b) The manually generated C code.

7.2 Creating Platforms in ESE FrontEnd

For testing various platforms, we have created three types of platforms for all applications. One is the simplest platform which has only one processor where all processes are mapped into. Another platform has two processors, one shared bus, and a transducer (TX) [5]. Note that, in these two platforms, all the channels among processes mapped to the same processor are implemented by using double-handshake (DH) channels instead of the original Kahn channel which is being developed. Therefore, all processes in the same processor communicate each other via the DH channels, thus after a

process writes a data, it should wait until another process reads the data. It can cause a deadlock problem if many channels are involved. Therefore, in this report, we assume that the input process network model to ESE FrontEnd has no deadlock problem by the DH channels, which guarantees the correct behavior of the generated TLMs. On the other hand, we use the TX for the communication between two processors since it supports the features like the Kahn channel except non-blocking write. The other is the platform where each process is mapped into one processor, all processes share a bus and they communicate via a TX each other. Note that the platforms created in this report are not optimal and the purpose of this chapter is to test those various platforms with various examples in ESE FrontEnd.

ESE FrontEnd provides several processing elements such as MicroBlaze processor and HWs and communication elements such as TX and buses. All the platforms have been created manually using the Graphical User Interface (GUI) which is provided by ESE FrontEnd.

Figure 9 shows the single processor platforms for H.264 encoder, decoder, and codec, respectively. Figure 10 shows the second platform which has two processors and one shared bus. In Figure 10, mapping the application to the platform is done manually by user based on a simple profiling. Finally, Figure 11 shows the last platform for all the applications. H.264 encoder has 13 processes, thus its platform in Figure 11 has 13 processors. For H.264 decoder, its platform has 12 processors. And the platform for H.264 codec has 25 processors. In these platforms, all processors share one bus and communicate via a TX each other.

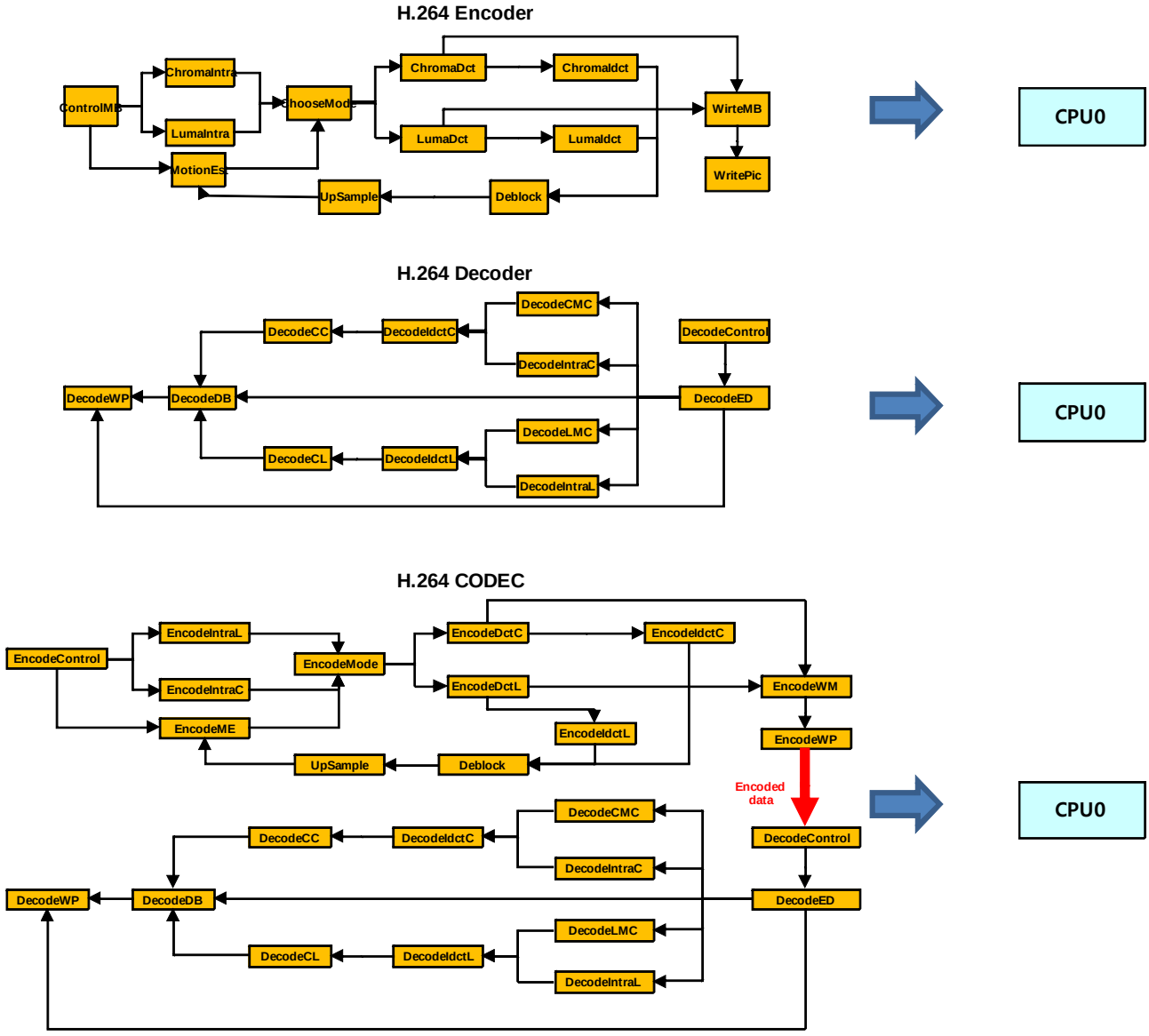


Fig 9. The single processor platforms for H.264 encoder, decoder and codec.

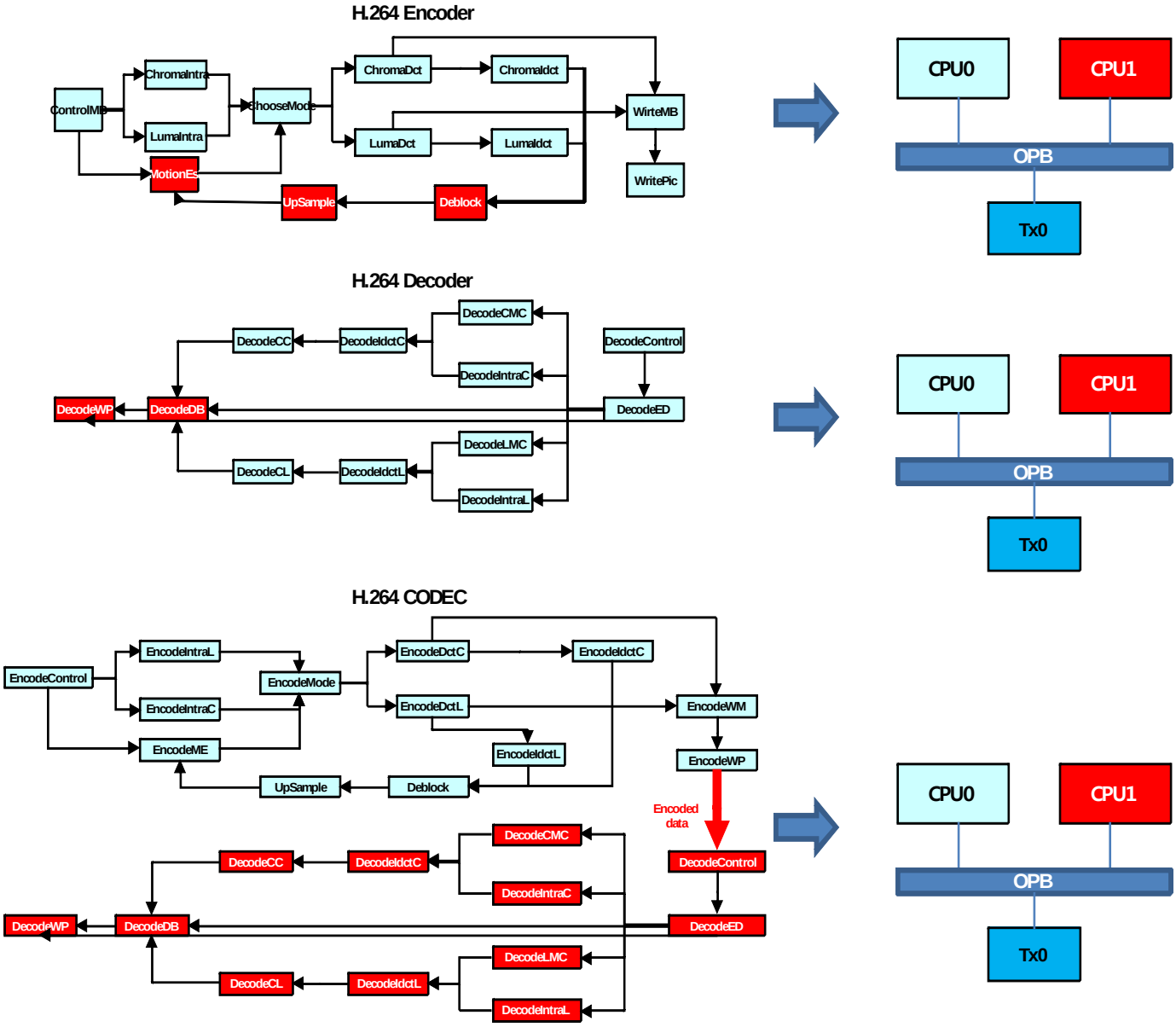


Fig 10. The two processor and one bus platforms for H.264 encoder, decoder and codec.

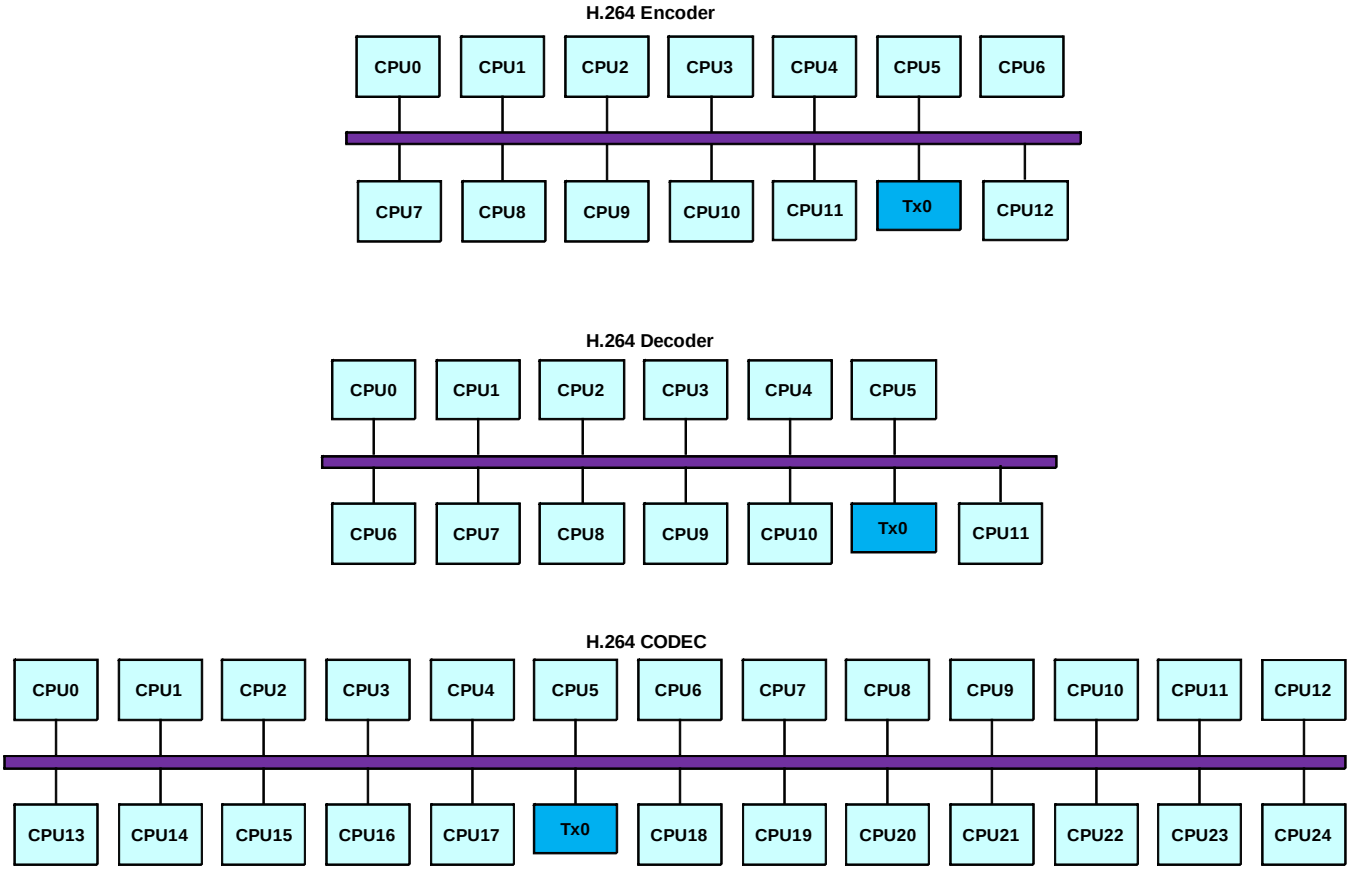


Fig 11. The platforms with a transducer for H.264 encoder, decoder and codec.

7.3 TLM Generation and Simulation

ESE FrontEnd can generate two types of TLMs. One is functional TLM which is for verifying its behavior and the other is timed TLM which is for performance estimation of the system [6]. The timed TLMs have real-time OS (RTOS) when more than two processes exist in the same processor. ESE FrontEnd provides the fully automated tool set to users. Therefore, we can easily generate the TLMs using ESE FrontEnd just by clicking ‘Generate Functional TLM’ or ‘Generate Timed TLM’ menu. And then, we can

simulate it just by clicking ‘Simulate Functional TLM’ or ‘Simulate Timed TLM’ menu in ESE FrontEnd.

Chapter 8 :Experimental Results

8.1 The Experimental Results on the Process Network Models by Hand

To show the quality of the manually generated process network models, we have measured their simulation times and code sizes. Table 1 shows the simulation time in seconds and the code size in lines for H.264 encoder, H.264 decoder and H.264 codec respectively. All of three models are encoding and decoding 15 frames whose format is QCIF on a 2GHz Linux machine. As shown in Table 1, H.264 decoder has less code size and faster simulation speed than H.264 encoder. This is because H.264 decoder does not have the motion estimation.

Table 1 The simulation time and codes size of the manually generated SystemC model

	Simulation time (sec.)	Code size (# of lines)
H.264 encoder	2.10	20.5 K
H.264 decoder	0.59	11.3 K
H.264 codec	2.32	31.2 K

8.2 The Experimental Results on the TLMs Generated by ESE

We have also measured the simulation time and code size of the TLMs generated by ESE FrontEnd. Table 2 shows the simulation time of the TLMs for H.264 encoder, decoder and codec respectively. It should be reminded that all the channels between processes in the same processor are generated by using DH channels. It is because FIFO

channel in ESE RTOS model is under development and is not yet available. Therefore, direct comparison with the manual version is not possible. However, we can still compare the simulation time of the manual version with that of the functional TLMs for Platform 1 generated by ESE. Platform 1 has no communication between processes. The only difference between the manually written process network models and the automatically generated TLMs by ESE FrontEnd is that they are using different channel primitives. As shown in Table 2, the simulation time of the functional TLMs for Platform 1 is fast enough even comparing to that of the manually created one. It is interesting that the simulation time keeps increasing as the platform becomes more complicated. It is because the generated models should have more information on the system architecture, such as bus and TX [7]. This results in more events that need to be simulated, thus the simulation speed becomes slow. For the timed TLMs, it takes more time to complete the simulation. It is natural because they have an overhead to run RTOS scheduling and accumulate timing delay for each basic blocks in the application source codes.

Table 2 The simulation time of the generated TLMs

	H.264 Encoder		H.264 Decoder		H.264 CODEC	
	Functional TLM	Timed TLM	Functional TLM	Timed TLM	Functional TLM	Timed TLM
Platform1	3.73	32.17	1.22	10.73	4.81	49.94
Platform2	4.12	50.01	1.25	13.31	4.85	83.86
Platform3	21.83	57.01	7.29	17.72	64.37	115.45

Table 3 shows the code size of the generated TLMs for H.264 encoder. We have measured the code size by part to clearly compare it to the code size in Table 1. For the

functional TLMs, the first column shows the size of the input C code, the second column shows the size of the newly generated codes which include bus model, TX model, etc., and the third column shows the total code size. Platform 1 has 20.1 K lines of code, which is similar to the manually generated process network model in Table 1. Like the simulation time, the code size is increasing as the platform gets more complicated. It is due to more features such as bus model, TX model and RTOS model. For the timed TLMs, the first column shows the size of the input C code like the functional models. The second column shows the annotated code size for the time information. The third column shows the size of the newly generated code. The forth column shows the total code size. The fifth column shows the code size increase in percentage comparing to the functional TLMs and the last column shows the increase of the total size without the annotated code. In the fifth column, the increase is mainly due to the code for annotating the timing delay for performance statistics. However, such a code is not really used in implementing the system at Register Transfer Level (RTL) or board level. Therefore, we have measured the increase of the total code size except the annotated code size in the last column. The amount of the increase is at most 10.9 % in Platform 3.

Table 3 The code size of the generated TLMs for H.264 Encoder

	H.264 Encoder								
	Functional TLM			Timed TLM					
	Input C code (lines)	Generated code (lines)	Total (lines)	Input C code (lines)	Time annotation in C code (lines)	Generated code (lines)	Total (lines)	Increase of total code size (%)	Increase except annotated code (%)
Platform1	19.0K	1.1K	20.1K	19.0K	6.7K	2.3K	29.0K	44.2	5.9
Platform2	19.0K	2.8K	21.8K	19.0K	6.8K	4.6K	30.4K	39.4	8.2
Platform3	19.0K	9.4K	28.4K	19.0K	6.8K	12.5K	38.3K	34.8	10.9

Table 4 and Table 5 show the code size of the generated TLMs for H.264 decoder and H.264 codec respectively. Like H.264 encoder, the functional TLM for Platform 1 has the similar code size compared to the manually written process network model. And the increase of the total code size for the timed TLMs is similar to that of H.264 encoder comparing to the functional TLMs. The increase of the total code size without the timing annotation part is also similar to that of H.264 encoder. As a result, it can be concluded that, for all the examples, the average increase is only about 9 %.

Table 4 The code size of the generated TLMs for H.264 Decoder

	H.264 Decoder								
	Functional TLM			Timed TLM					
	Input C code (lines)	Generated code (lines)	Total (lines)	Input C code (lines)	Time annotation in C code (lines)	Generated code (lines)	Total (lines)	Increase of total code size (%)	Increase except annotated code (%)
Platform1	10.4K	0.8K	11.2K	10.4K	4.5K	1.7K	16.6K	48.2	8.0
Platform2	10.4K	2.8K	13.2K	10.4K	4.6K	4.3K	19.3K	46.2	10.1
Platform3	10.4K	5.7K	16.1K	10.4K	4.6K	7.9K	22.9K	42.2	13.6

Table 5 The code size of the generated TLMs for H.264 codec

	H.264 CODEC								
	Functional TLM			Timed TLM					
	Input C code (lines)	Generated code (lines)	Total (lines)	Input C code (lines)	Time annotation in C code (lines)	Generated code (lines)	Total (lines)	Increase of total code size (%)	Increase except annotated code (%)
Platform1	29.2K	1.7K	30.9K	29.2K	11.4K	3.7K	44.3K	43.3	6.4
Platform2	29.2K	2.7K	31.9K	29.2K	11.4K	5.0K	45.6K	42.9	7.2
Platform3	29.2K	14.7K	43.9K	29.2K	11.4K	19.7K	60.5K	37.8	11.4

Finally, we show the code productivity gain obtained by the automation. Table 6, 7, and 8 shows the productivity gain for H.264 encoder, decoder and codec respectively. We have calculated the estimated manual time assuming that the man productivity per day is 30 lines. As shown in the tables, the productivity gain obtained by the automated ESE is very high for both the function TLMs and the timed TLMs.

Table 6 Productivity gain by automation for H.264 encoder

	H.264 Encoder							
	Functional TLM				Timed TLM			
	Generated Code (lines)	Estimated manual time (day)	Automatic generation time (sec.)	Productivity gain	Generated Code (lines)	Estimated manual time (day)	Automatic generation time (sec.)	Productivity gain
Platform1	1.1K	37	0.3	4.5×10^5	2.3K	76	20.9	1.3×10^4
Platform2	2.8K	93	0.4	8.3×10^5	4.6K	153	22.9	2.4×10^4
Platform3	9.4K	313	0.8	1.4×10^6	12.5K	416	23.7	6.3×10^4

Table 7 Productivity gain by automation for H.264 decoder

	H.264 Decoder							
	Functional TLM				Timed TLM			
	Generated Code (lines)	Estimated manual time (day)	Automatic generation time (sec.)	Productivity gain	Generated Code (lines)	Estimated manual time (day)	Automatic generation time (sec.)	Productivity gain
Platform1	0.8K	27	0.2	4.8×10^5	1.7K	57	13.3	1.5×10^4
Platform2	2.8K	93	0.3	1.1×10^6	4.3K	143	15.3	3.4×10^4
Platform3	5.7K	190	0.5	1.4×10^6	7.9K	263	15.5	6.1×10^4

Table 8 Productivity gain by automation for H.264 codec

	H.264 CODEC							
	Functional TLM				Timed TLM			
	Generated Code (lines)	Estimated manual time (day)	Automatic generation time (sec.)	Productivity gain	Generated Code (lines)	Estimated manual time (day)	Automatic generation time (sec.)	Productivity gain
Platform1	1.7K	57	0.5	4.1×10^5	3.7K	123	31.5	1.4×10^4
Platform2	2.7K	90	0.7	4.6×10^5	5.0K	167	33.6	1.8×10^4
Platform3	14.7K	490	1.9	9.3×10^5	19.7K	657	35.8	6.6×10^4

We have successfully generated all the TLMs for all the examples, simulated them by using ESE FrontEnd. As shown in the experimental results, we can significantly reduce the design time by using the fully automated ESE FrontEnd and we can also know that ESE FrontEnd is scalable enough to be used for efficient design space exploration for industrial scale applications.

8.3 Verifying the Functional Correctness

In order to verify the correctness of the behavior of all the process network models and TLMs, we use a software tool called ‘mplayer’ which is a free and open source media player and available at http://www.mplayerhq.hu. It can take a 264 or a YUV file as an input. The UNIX commands to run the 'mplayer' are as follows;

>>mplayer –fps 5 –fixed-vo –vo sdl –loop 1 encoded.264 ↵

>>mplayer –rawvideo on:w=176:h=144:fps=5:format=i420 –fixed-vo –vo gl decoded.yuv ↵

Figure 12 shows the snapshots of the output for H.264 codec.

Fig 12. The snapshots of the output for H.264 codec.

Chapter 9 :Conclusion

In this report, we have explained how to create the process network models in SystemC from the original H.264 algorithm in C. And we have successfully generated and simulated TLMs using ESE FrontEnd from them. We hope this report helps someone who wants to start with ESE FrontEnd from process network model.

Bibliography

[1] G. Kahn, “The semantics of a simple language for parallel programming,” in *Proceedings of the IFIP Congress*, pp. 471-475, 1974.

[2] T. Basten and J. Hoogerbrugge, *Efficient execution of process networks*. Communication Process Architectures, IOS Press, 2001.

[3] SystemC Intuitive. *http://www. systemc.org.*

[4] ITU-T, ISO/IEC JTC1, “Advanced video coding for generic audiovisual services,” ITU-T Recommendation H.264-ISO/IEC 14496-10 AVC, 2003.

[5] H. Cho, S. Abdi, and D. Gajski, “Interface synthesis for heterogeneous multi-core systems from transaction level models,” *Language, Compiler and Tool Support for Embedded Systems*, pp. 140-142, 2007.

[6] Y. Hwang, S. Abdi, and D. Gajski, “Cycle-approximate retargetable performance estimation at the transaction level,” Design, Automation & Test in Europe, pp. 3-8, 2008.

[7] L. Yu, S. Abdi, D. Gajski, “Transaction level platform modeling in SystemC for multiprocessor designs”, Technical Report TR07-01, UC Irvine, 2007.

Appendix A: The Top SystemC Code for The Process Network Model of H.264 Codec

```
//TopCodec.cpp for H.264 Codec - 2008.3      by Yongjin Ahn

#include "systemc.h"
#include "EncodeME.h"
#include "EncodeDB.h"
#include "EncodeUS.h"
#include "EncodeMode.h"
#include "EncodeDctL.h"
#include "EncodeDctC.h"
#include "EncodeIdctL.h"
#include "EncodeIdctC.h"
#include "EncodeIntraL.h"
#include "EncodeIntraC.h"
#include "EncodeWP.h"
#include "EncodeWM.h"
#include "EncodeControl.h"
#include "DecodeControl.h"
#include "DecodeED.h"
#include "DecodeIntraL.h"
#include "DecodeIntraC.h"
#include "DecodeLumaMC.h"
#include "DecodeChromaMC.h"
#include "DecodeIdctL.h"
```

```
#include "DecodeIdctC.h"
#include "DecodeCL.h"
#include "DecodeCC.h"
#include "DecodeDB.h"
#include "DecodeWP.h"

int sc_main(int, char **) {

        //encode blocks
        EncodeControl *encodecontrol;
        EncodeWP *encodewp;
        ……
        ……

        EncodeME *encodeme;
        EncodeDB *encodedb;

        //decode blocks
        DecodeControl    *decodecontrol;
        DecodeED         *decodeed;
        ……
        ……
        DecodeDB         *decodedb;
        DecodeWP *decodewp;

        //encode blocks
        encodecontrol = new EncodeControl("encodecontrol");
        encodewp = new EncodeWP("encodewp");
        ……
```

```
……
encodeme = new EncodeME("encodeme");
encodedb = new EncodeDB("encodedb");

//decode blocks
Decodecontrol = new DecodeControl("decodecontrol");
decodeed= new DecodeED("decodeed");
……
……
decodedb= new DecodeDB("decodedb");
decodewp                = new DecodeWP("decodewp");

//encode channels
sc_fifo <mbimgy> R3_1("R3_1", 1);
sc_fifo <mbimgy> R3_2("R3_2", 1);
sc_fifo <mbimgy> R3_3("R3_3", 1);
sc_fifo <mbimguv> R4_1("R4_1", 1);
sc_fifo <mbimguv> R4_2("R4_2", 1);
……
……
……
sc_fifo <mbimgy> R64("R64", 1);
sc_fifo <mbrefpicnum> R65("R65", 1);
sc_fifo <allmv> R66("R66", 1);
sc_fifo <allmv> R67_1("R67_1", 1);
sc_fifo <allmv> R67_2("R67_2", 1);
sc_fifo <int> R68("R68", 1);
sc_fifo <foury> R69("R69", 1);
sc_fifo <refeleven> R70("R70", 1);
```

```
sc_fifo <Bitstream> R71("R71", 1);
sc_fifo <Macroblock> R72("R72", 1);

//decode channels
sc_fifo <dec_StorablePicture>    DR4("DR4",1);
sc_fifo <TransImage>    DR5("DR5",1);
sc_fifo <dec_seq_parameter_set_rbsp_t>    DR6("DR6",1);
sc_fifo <dec_ImageParameters>    DR7("DR7",1);
……
……
……
sc_fifo <mbry>    DR84("DR84",1);
sc_fifo <mbry>    DR85("DR85",1);
sc_fifo <mbimgy4>    DR86("DR86",1);
sc_fifo <mbimgy4>      DR87("DR87",1);
sc_fifo <imgydata>    DR88("DR88",1);
sc_fifo <imguv2data>    DR89("DR89",1);
sc_fifo <dec_StorablePicture>    DR90("DR90",1);
sc_fifo <imgydata>    DR92("DR92",1);
sc_fifo <imguv2data>    DR94("DR94",1);

//encode --> decode
sc_fifo <E2D_data>    ED1("ED1",1);
sc_fifo <int>    ED2("ED2",1);

//encode interconnections
encodecontrol->out_mb_imgY_org (R3_1);
encodeme->in_MB_imgY_org (R3_1);
encodecontrol->out_mb_imgY_org (R3_2);
```

encodeintral->in_mb_imgY_org (R3_2);

encodecontrol->out_mb_imgY_org (R3_3);

encodedctl->in_mb_imgY_org (R3_3);

encodecontrol->out_mb_imgUV_org (R4_1);

encodeintrac->in_mb_imgUV_org (R4_1);

encodecontrol->out_mb_imgUV_org (R4_2);

encodedctc->in_mb_imgUV_org (R4_2);

encodecontrol->out_img_number(R5_1);

encodeme->in_number(R5_1);

encodecontrol->out_img_number(R5_2);

encodewm->in_img_number(R5_2);

encodecontrol->out_img_type(R6_1);

encodeme->in_type(R6_1);

encodecontrol->out_img_type(R6_2);

encodewm->in_type(R6_2);

encodecontrol->out_img_type(R6_3);

encodemode->out_check_skip(R19);

encodewm->in_check_skip(R19);

encodemode->out_mpr(R20_1);

encodedctl->in_mpr(R20_1);

encodemode->out_mpr(R20_2);

encodeidctl->in_mpr(R20_2);

encodemode->out_b8pdir(R29);

encodewm->in_b8pdir(R29);

encodedctc->out_cr_cbp(R30);

encodewm->in_cr_cbp(R30);

encodedctc->out_cr_cbp_blk(R31);

encodedb->in_cr_cbp_blk(R31);

encodedctc->out_M7(R32);

encodeidctc->in_M7(R32);
encodedctc->out_img_cofAC_cr(R33);
encodewm->in_cofAC_cr(R33);
encodedctc->out_img_cofDC_cr(R34);
encodewm->in_cofDC_cr(R34);
encodedctl->out_img_cofAC(R35);
encodewm->in_cofAC(R35);
……
……
……
……
……
encodemode->in_me_min_cost(R61);
encodeme->out_me_best_mode(R62);
encodemode->in_me_best_mode(R62);
encodeme->out_img_all_mv(R67_1);
encodewm->in_img_all_mv(R67_1);
encodeme->out_img_all_mv(R67_2);
encodemode->in_img_all_mv(R67_2);
encodeme->out_list_size(R68);
encodewm->in_list_size(R68);
encodeus->out_4Y(R69);
encodeme->in_imgY_ups(R69);
encodeus->out_ref11(R70);
encodeme->in_imgY_11(R70);
encodewm->out_bitstream(R71);
encodewp->in_bitstream(R71);
encodecontrol->out_currMB(R72);
encodewm->in_currMB(R72);

```
//decode interconnections
decodecontrol->out_con_ed_picture(DR4);
decodedb->in_df_picture(DR4);
decodecontrol->out_con_ed_sps(DR6);
decodewp->in_wp_sps(DR6);
decodecontrol->out_con_ed_image(DR7);
decodeed->in_ed_image(DR7);
decodecontrol->out_con_ed_const_intra(DR8);
decodeed->in_ed_constraint_intra(DR8);
decodecontrol->out_con_ed_lumaintrascale(DR9);
decodeed->in_ed_luma_intra(DR9);
decodecontrol->out_con_ed_chromaintrascale(DR10);
decodeed->in_ed_chroma_intra(DR10);
decodecontrol->out_con_ed_lumainterscale(DR11);
decodeed->in_ed_luma_inter(DR11);
decodecontrol->out_con_ed_chromainterscale(DR12);
decodeed->in_ed_chroma_inter(DR12);
decodeed->out_ed_df_image(DR36);
decodedb->in_df_image(DR36);
decodeed->out_ed_cd_mb_type(DR47);
decodeidctc->in_cd_mb_type(DR47);
……
……
……
……
……
decodeidctc->in_cd_interimguv(DR83);
decodeidctl->out_ld_residuey(DR84);
```

```
        decodecl->in_clm_imgy(DR84);
        decodeidctc->out_cd_residueuv(DR85);
        decodecc->in_clm_imguv(DR85);
        decodecl->out_clm_imgy(DR86);
        decodeintral->in_lip_imgy(DR86);
        decodecc->out_clm_imguv(DR87);
        decodeintrac->in_cip_imguv(DR87);
        decodecl->out_clm_ydata(DR88);
        decodedb->in_clm_ydata(DR88);
        decodecc->out_clm_uvdata(DR89);
        decodedb->in_clm_uvdata(DR89);
        decodedb->out_df_picture(DR90);
        decodewp->in_wp_picture(DR90);
        decodewp->out_wp_listluma(DR92);
        decodelumamc->in_lm_list(DR92);
        decodewp->out_wp_listchroma(DR94);
        decodechromamc->in_cm_list(DR94);

        //encode --> decode
        encodewp->out_enc2dec(ED1);
        decodecontrol->in_enc2dec(ED1);
        encodewp->out_enc2dec_size(ED2);
        decodecontrol->in_enc2dec_size(ED2);

        sc_start(-1);
        return 0;
};
```