Chapter 4 H.264 Decoder Implementation and Optimization

In this chapter, we will describe an H.264 baseline decoder implementation and optimization on DSP. We will describe how to optimize C/C++ code based on DSP architecture and how to optimize the H.264 baseline decoder based on VP[3.]

4.1 Profile of H.264 decoder on DSP

We do some modification on the H.264 decoder C source code, and then implement this code on DSP. After profiling this code by TI CCS profiler, we optimize the most computationally heavy parts of these modified codes. We choose QCIF as our test format. The test sequence is the "foreman" sequence and the length of this sequence is 10 frames with 5 Intra periods. Fig 4-1 shows the program flow of the H.264 decoder reference software in version JM 7.3.

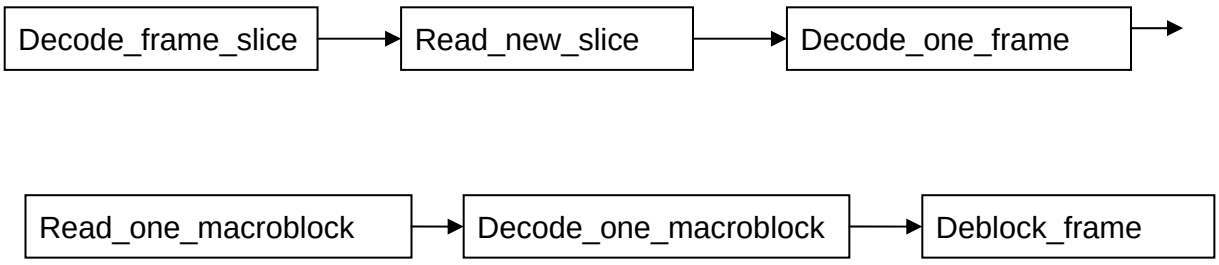

Fig 4-1 Decoder program flow

Table 4-1 shows the clock cycle of the most complex functions. Fig 4-2 shows the clock cycle distribution of these functions without optimization. We find that the *"decode_one_macroblock"* and *"read_one_macroblock"* require 54% and 28% of

the total clock cycle. Hence we focus on the optimization of these two functions first.

Table 4-1 Clock cycle of the most complexity function

QP28/Foreman	Clock Cycle	Percent (%)	NO. of Execution
decode_one_macroblock	**501835424**	**54**	**990**
read_one_macroblock	**259499384**	**28**	**990**
decblock_frame	**32848668**	**4**	**10**
Total	**926511674**	**100**	**1**

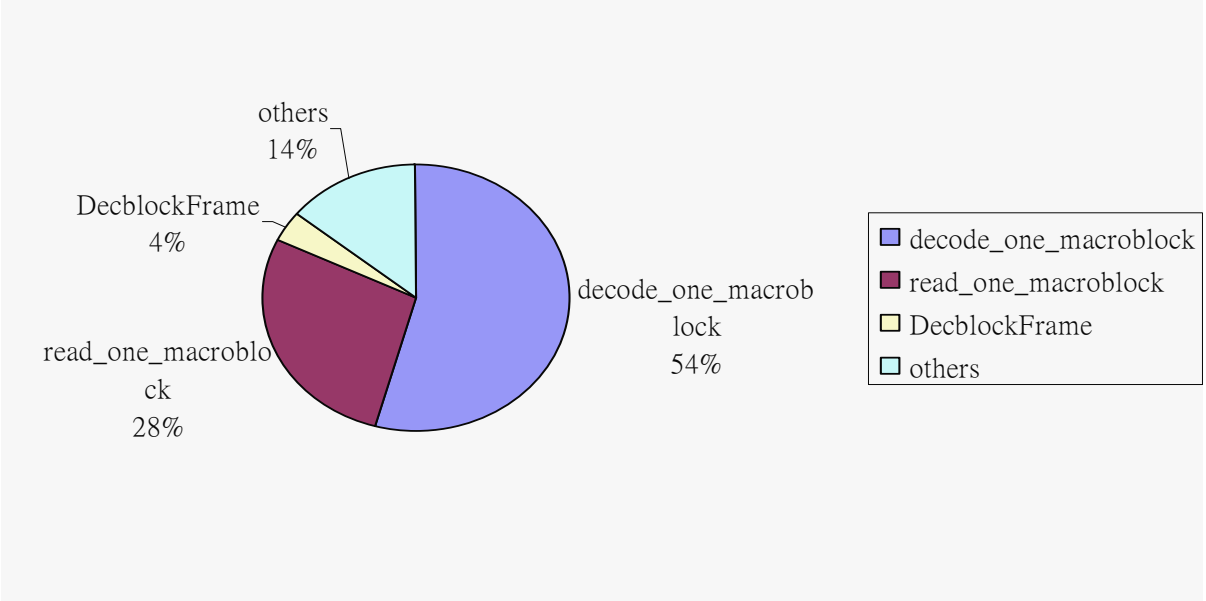

Fig 4-2 Distribution of clock cycle of each function

"*Decode_one_macroblock*" is the function that performs motion compensation or intra prediction. This function contains several sub function. We will describe each function as follows.

"*get_block*": Get motion compensated block.

"*itrans*": Inverse Discrete Cosine Transform.

"*intrapred*": Makes and returns 4×4 blocks with all 5 intra prediction modes

"*intrapred_chroma*": Makes and returns 4×4 chroma blocks

"intrapred_luma 16X16" : Makes and returns16×16 Luma blocks

Table 4-2 shows the sub function and their clock cycles.

Table 4-2 Profile of the H.264 Decoder : decode_one_macroblock

QP28/Foreman	Clock Cycle	Percent (%)	NO. of Execution
Get_Block	**273905340**	**55**	**12656**
Itrans	**95182560**	**19**	**23760**
Intrapred	**14240186**	**3**	**2992**
Intrapred_Chroma	**3423458**	**0.7**	**398**
Intrapred_Luma16X16	**230483**	**0.04**	**12**
Total	**501835424**	**100**	**990**

"Read_one_macroblock" is the function used for Entropy decoding. We describe this function below.

"ReadCBPandCoeffsFromNAL": Get coded block patterns and coefficients from the NAL

"ReadMotionInfoFromNAL": Get motion information from NAL.

"Read_ipred_modes": Get intra prediction mode.

Table 4-3 shows the sub-functions of *"read_one_macroblock"* and their clock cycles.

Table 4-3 Profile of the H.264 Decoder : read_one_macroblock

QP28/Foreman	Clock Cycle	Percent (%)	NO. of Execution
ReadCBPandCoeffsFromNAL	**230096749**	**89**	**752**
ReadMotionInfoFromNAL	**11071392**	**4.3**	**553**
Read_ipred_modes	**6447989**	**2.5**	**752**
Total	**259499384**	100	990

4.2 Optimize C/C++ Code

In this section, we will describe several methods that we can use to optimize our C/C++ code.

4.2.1 Setting of CCS Compiler

Code Composer Studio (CCS) is a useful GUI tool that helps us to develop DSP codes. CCS can compile the C codes and assembles them it into the COFF file format. to generate assembly codes efficiently. Table 4-4 shows the compiler options for improving performance whereas Table 4-5 shoes those options that had better be avoided.

Table 4-4 Compiler options for performance [16]

Option	Description
–o3†	Represents the highest level of optimization available. Various loop optimizations are performed, such as software pipelining, unrolling, and SIMD. Various file level characteristics are also used to improve performance.
–pm‡	Combines source files to perform program–level optimization.
–op2	Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler. This improves variable analysis and allowed assumptions.
–oi0	Disables all automatic size–controlled inlining, (which is enabled by –o3). User specified inlining of functions is still allowed.
–ms2–ms3	Optimizes primarily for code size, and secondly for performance.

Table 4-5 Compiler options that had better be avoided [16]

Option	Description
–g/–s/ –ss/–mg	These options limit the amount of optimization across C statements leading to larger code size and slower execution.
–mu	Disables software pipelining for debugging. Use –ms2/–ms3 instead to reduce code size which will disable software pipelining among other code size optimizations.
–o1/–o0	Always use –o2/–o3 to maximize compiler analysis and optimization. Use code size flags (–ms*n*) to tradeoff between performance and code size.
–mz	Obsolete. On pre–3.00 tools, this option may have improved your code, but with 3.00+ compilers, this option will decrease performance and increase code size.

4.2.2 Software Pipelining

Software pipelining is a technique used to schedule instructions in loop so that multiple iterations of the loop can be executed in parallel. Software pipelining is to implements parallel instructions, fills delay slots with useful instructions, unrolls loops and maximizes the usage of functional units. This technique is a useful way to improve performance. When the compiler options –o2 or –o3 are used, the compiler will gather information from the program for the optimization process. As more information is gathered, better results may be obtained. Listed below are some kinds of information that can be provided by the programmer to help the optimization process.

Trip Count

A trip count is the number of loop iterations that need to be executed. If the compiler can guarantee that at least n loops will be executed, then n is the known minimum trip count. The programmer can use the MUST_ITERATE pragma to provide this information to compiler. It can reduce code size by preventing the generation of redundant loops. If the programmer can provide the trip count information to compiler, the compiler can generate faster and more compact code. The syntax of the of the MUST-ITERATE pragma is :

#pragma MUST_ITERATE (min, max, multiple);

The arguments min and max are programmer-guaranteed minimum and maximum trip counts. The trip count is the number of times a loop iterates. The trip count of the loop must be evenly divisible by the “multiple”.

Loop Unrolling

Another way to improve the performance is to unroll the loop. Loop unrolling is to expand small loops so that every iterations of the loop appear in your codes. This optimization will increase the number of instructions available for parallel computing. There are three ways loop unrolling can be performed:

1. The compiler may unroll the loop automatically.
2. Programmer can unroll the loop using the UNROLL pragma
3. Programmer can unroll the C/C++ code by himself.

The syntax of the UNROLL pragma is :

#pragma UNROLL (n);

Software pipelining can improve the performance significantly. However, the compiler will not perform software pipelining whenever any of the following conditions happens: [16]

1. If a register value lives too long, the code is not software-pipelined.
2. If the loop has complex condition codes within the body that require more than five condition registers, the loop is not software pipelined.
3. Although a software-pipelined loop can contain intrinsics, it cannot contain function calls, including codes that calls the run-time support routines.
4. The loop contains conditional breaks.
5. In a nested loops, the innermost loop is the only one that can be software-pipelined.

4.2.3 Using Intrinsics

The C6000 compiler provides intrinsics, which are special functions that map directly to inlined C64x instructions, to optimize C/C++ code efficiently. The intrinsic

functions are optimized codes based on the knowledge and techniques of DSP architechture. They can be recognized by the TI CCS compiler only. The intrinsic functions are specified with a leading underscore (_). Fig 4-3 shows a parts of the intrinsic function for C6000 series DSP.

C Compiler Intrinsic	Assembly Instruction	Description	Device
int **_max2** (int *src1*, int *src2*); int **_min2** (int *src1*, int *src2*); unsigned **_maxu4** (uint *src1*, uint *src2*); unsigned **_minu4** (uint *src1*, uint *src2*);	**MAX2** **MIN2** **MAXU4** **MINU4**	Places the larger/smaller of each pair of values in the corresponding position in the return value. Values can be 16–bit signed or 8–bit unsigned.	C64x
ushort & **_mem2**(void *ptr);	**2 LDB /** **2 STB**	Allows unaligned loads and stores of 2 bytes to memory.	

Fig 4-3 Intrinsic functions of the TI C6000 series DSP (part.) [11]

4.2.4 Packed data Processing

The C64x DSP is a 32-bit fixed-point processor, which is for 32-bit data operation. In order to maximize data throughput, it is prefer to use single load or store to access multiple data values consecutively located in memory. For example, if we can place four 8-bit data or two 16-bite data in a 32-bit register, we can do more than one operation in one cycle. This process can improve the code efficiency and performance significantly.

4.2.5 Memory Usage Strategy

As mentioned in Section 3.2.2, the size of the internal memory of C64X is 256K bytes. However, when developing the codes, the program may require lager memory size than the internal memory. When decoding one frame, we cannot load all data into the internal memory and some of this data will be access many times. For this reason, data which are accessed less frequently will be put into the external memory while others remain in the internal memory. We can use the pragma DATA_SECTION to allocate more importance data into the internal memory. The DATA_SECTION

pragma is useful if you have data objects that you want to link into an area separate from the .bss section. This directive is illustrated in the following example.

```
#pragma DATA_SECTION (buf,"Tao_sect.");
Int buf[100];
```

where "*buf*" is the buffer we declared in your C/C++ program, "*Tao_sect.*" is the section name which can be allocated into the desired memory section in the CMD file.

The above pragma is for data allocation. There is another pragma CODE_SECTION that can allocate the program code section into the desired memory. The CODE_SECTION pragma is useful if you want to link some code objects into an area separate from the .text section. The syntax of the CODE_SECTION pragma is expressed as:

```
#pragma CODE_SECTION (text,"Tao_sect.");
Void text( ){
/*foo code*/
}
```

where "text" is the function name, "Tao_sect." is the section name which can be allocated into the desired memory section in the CMD file.

4.2.6 Optimization Results

In this section, we compare the optimized functions with the original ones. In the reference software of the H.264 decoder, there are three major functions: read_one_macroblock, decode_one_macroblock, and decblock_frame. Table 4-6 shows the optimization results of some major functions. From these results, we find that the most dominating function has changes. Before optimization, the most time-consuming function is "*decode_one_macroblock*". After the optimization, "*read_one_macroblock*" becomes the most dominating function. We also show in Table 4.7 ~Table 4.10 the code size and clock cycles of the sub-functions in some major functions.

Table 4-6 clock cycle of the H.264 decoder

QP28/Foreman	Non-optimized	Optimized	Ratio
decode_one_macroblock	501835424	72023154	6.96
read_one_macroblock	259499384	195424945	1.32
decblock_frame	32848668	13003263	2.5
Total	926511674	377102400	2.5

Table 4-7 clock cycle of the H.264 decoder : decode_one_macroblock

QP28/Foreman	Non-optimized (cycle)	Optimized (cycle)	Ratio
Get_Block	273905340	21645906	12.7
Itrans	95182560	8458560	11.2
Intrapred	14240186	8913111	1.6
Intrapred_Chroma	3423458	1640386	2.1
Intrapred_Luma16X16	230483	93745	2.5

Table 4-8 code size of the H.264 decoder :decode_one_macroblock

QP28/Foreman	Non-optimized (code size)	Optimized (code size)	Ratio
Get_Block	7752	3861	2.1
Itrans	1700	356	4.8
Intrapred	9276	5688	1.6
Intrapred_Chroma	11055	1825	6.1
Intrapred_Luma16X16	31429	4733	6.6

Table 4-9 clock cycle of the H.264 decoder: read_one_macroblock

QP28/Foreman	Non-optimized (cycle)	Optimized (cycle)	Ratio
ReadCBPandCoeffsFromNAL	230096749	180026051	1.28
ReadMotionInfoFromNAL	11071392	6422894	1.7
Read_ipred_modes	6447989	3778452	1.7

Table 4-10 code size of the H.264 decoder: read_one_macroblock

QP28/Foreman	Non-optimized (code size)	Optimized (code size)	Ratio
ReadCBPandCoeffsFromNAL	41107	17194	2.4
ReadMotionInfoFromNAL	10966	13660	0.8
Read_ipred_modes	1808	2840	0.6

4.3 Implementation On DSP

In this section, we implement H.264 decoder on VP[3]. We'll first implement the H.264 over a single DSP. Then, due to the fact that VP3 possesses 8 DSPs, we also practice the implementation H.264 on two DSPs as parallel operation. The implementation details and experiment results are described as follow.

4.3.1 Over a Single DSP

In this section, we implement an H.264 decoder on one DSP. Fig 4-5 shows the system flowchart and communication interface between the host and the DSP. There is a buffer for receiving bitstream data sent from host. We design a mechanism to avoid buffer overflow or underflow. This mechanism contains two parts. The first part is that when the buffer pointer points to the center of the buffer that means the upper side of the buffer has been used and we can refresh this region. Hence, DSP will send an interruption to host. When host receive the interruption, it will send bitstream in

the size of half of the buffer size to the upper side of DSP buffer. The second part is that when the buffer pointer points to the end of the buffer that means the lower side of the buffer has been used. DSP also send an interrupt to host. When host receive interruption, it will send bitstream to the lower side of the DSP buffer. After one cycle, this buffer is replaced by the new data bitstream and we can avoid any overflow or underflow during decoding processing. In the following, we describe the system flow chart step by step.

STEP.1 The host downloads program to the DSP and The program starts.

STEP.2 The host sends a video bitstream to the DSP n the size of half the DSP buffer and the DSP starts to decode.

STEP.3 The host waits for the information that indicates the completion of decoding or the request of new data.

STEP.4 The DSP will check the buffer status. If the buffer is OK, then go Step 5, otherwise, send interruption to the host torequir new bitstream. Then go to Step 5.

STEP.5 When the DSP completely decodes one frame complete, it sends an interruption to the host. Then, the host will require frame datafrom the DSP and output as a file.

STEP.6 If all the bitstreams are decoded, end. Otherwise go back to Step 3.

4.3.1.1 Experiment Results of the Whole System

In this section, we consider the overall performance of the completely implemented H.264 decoder on one DSP. The decoding speed of the system depends on the quantization parameter. The test sequence is the QCIF “foreman” sequence with 30 frames. When the quantization parameter increases, the bitstream size increases. Furthermore, the entropy decoding speed depends on the bitstream size. Table 4-11 shows the relation between clock cycle and bitstream size (quantization parameter). When QP increases, the bitstream size decreases.The decoding time of the function “*decode_one_macroblock*” is about the same, while the decoding time of the

function *"read_one_macroblock"* is decreasing. Fig 4-4 shows relationship between the number of clock cycles and QP. Table 4-12 shows the overall decoding speed over a single DSP.

Table 4-11 Relation between bitstream size and function

	Clock cycle			
QP	Decode_one_macroblock	Read_one_macroblock	Total	Bitstream size (KB)
16	**71885638**	**809615734**	**1005066498**	**43.2**
20	**71586198**	**524753243**	**713835469**	**25.8**
24	**71933634**	**324011601**	**508493796**	**15.1**
28	**72023154**	**195424945**	**377102400**	**9**
32	**73186106**	**134907576**	**316306899**	**5.63**
35	**73572604**	**107611246**	**288909132**	**4.1**
40	**71416524**	**73115402**	**252126190**	**2.56**
42	**66334579**	**64882945**	**238980866**	**2.18**

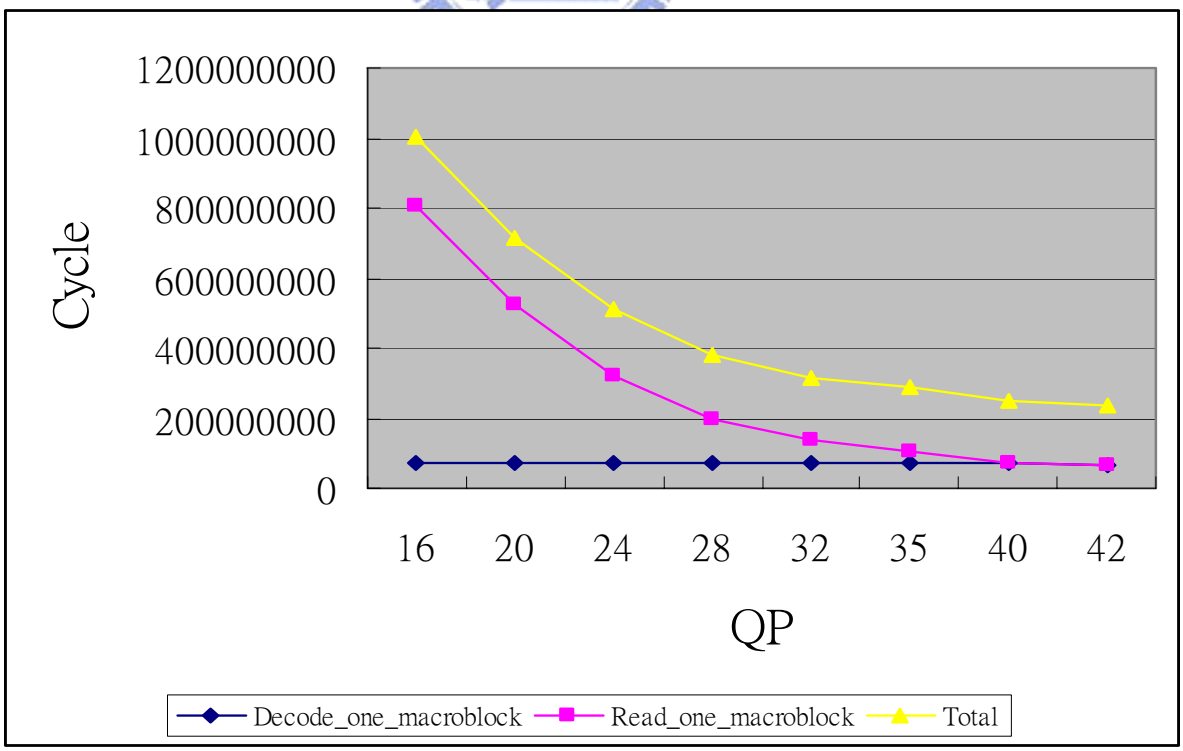

Fig 4-4 clock cycle of different QP

Table 4-12 overall speed using optimization and no optimization

	Average QCIF frame per second		
QP	**Non-optimized**	**Optimized with software pipelining**	**Ratio**
16	**0.55**	**1.5**	**2.7**
20	**0.8**	**1.7**	**2.3**
24	**1.2**	**2.1**	**1.8**
28	**1.5**	**2.3**	**1.6**
32	**1.7**	**2.5**	**1.5**
35	**1.8**	**2.5**	**1.5**
40	**1.9**	**2.6**	**1.5**
42	**2**	**2.7**	**1.4**

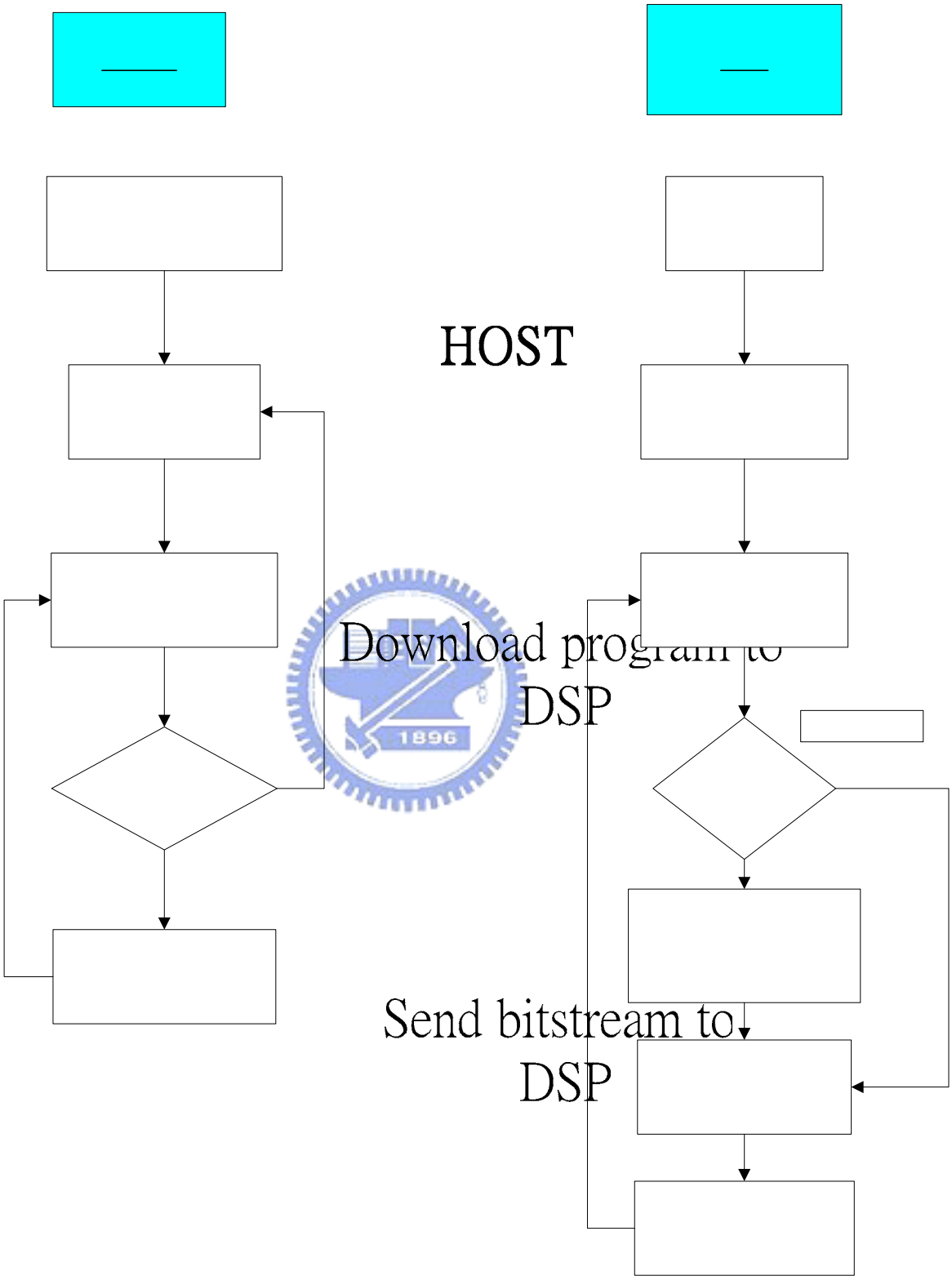

Fig 4-5 Flowchart for implementation over a single DSP

4.3.2 Over Two DSP's

In this section, we implement H.264 decoder on two DSPs. Considering the computation complexity and resource dependency, we implement entropy decoding on DSP 1 otherwise on DSP2. Fig 4-6 shows the system flow chart. We also employ the mechanism as described in section 4.2.1 to avoid buffer overflow or underflow. We describe the flow chart step by step.

STEP.1 The host downloads the program to DSP1 and DSP2. Then, the program starts.

STEP.2 The host sends a video bitstream in the size of half of the buffer size to DSP1

STEP.3 DSP1 starts to read NALU.

STEP.4 Check the buffer status. If the buffer status is OK, go to Step 5, otherwise, DSP1 send an interruption to the host to requir a new bitstream and then go to Step 5.

STEP.5 Entropy decoding.

STEP.6 Send the decoded data from DSP1 to DSP2.

STEP.7 DSP2 decodes one macroblock. If the current macroblock is at the end of frame, then DSP2 sends an interruption to the host otherwise return to step 6.

STEP.8 The host requires frame data from DSP2.

STEP.9 If the bitstream ends, decoding is over otherwise go to Step 3.

4.3.2.1 Experiment results of the whole system

In this section, we discuss the overall performance of the implemented H.264 decoder on two DSPs. Table 4-13 shows the overall decoding speed. After optimization, the frame rate can achieve 11~15 frames per second. There is a significant improvement in speed if compared with the implementation over a single DSP. The reason is that when we implement H.264 decoder on two DSPs, we can separate the code size and resources into two DSPs. Hence the codes and the important data can be put into the internal memory and the decoding speed increase significantly.

Table 4-13 Overall speed with optimization and without optimization

	Average QCIF frame per second		
QP	**Non-optimized**	**Optimized with software pipelining**	**Ratio**
16	**6.8**	**12**	**1.76**
20	**6.9**	**12**	**1.73**
24	**7**	**12.8**	**1.85**
28	**7.1**	**12.5**	**1.82**
32	**7.3**	**13.5**	**1.88**
35	**7.4**	**13.6**	**1.88**
40	**7.7**	**13.8**	**1.82**
42	**8.1**	**14.3**	**1.76**

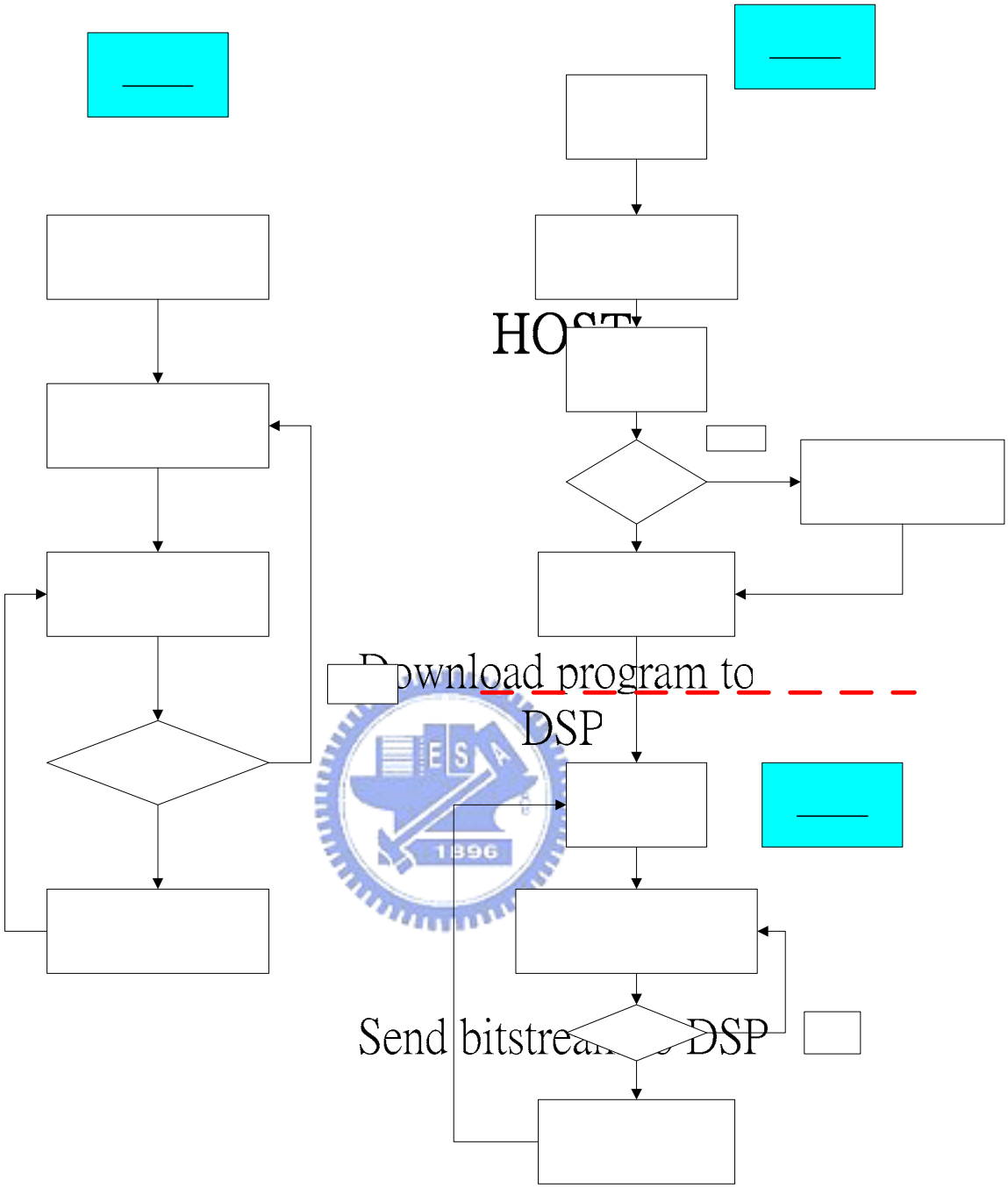

Fig 4-6 Flowchart for implementation over two DSP's

Wait for interruption

Yes

If interruption =1