Chapter 4

Experimental Results

In this chapter, we present the result of software and hardware which we proposed in chapter 2 and chapter 3. The number of instructions executed of interpolation procedure is demonstrated with and without our new instructions. One the other hand, the area overhead and timing issue of the hardware will be discussed later.

4.1 Software Performance

To make sure that our new instructions will be scheduled during compilation, we use *inline_assembly*. If the compiler parses a special keyword *asm*, it will call the assembler to parse the parenthetic codes. Because the compiler may or may not generate the best assembly code for procedures, our results are compared to a code which is written by us using *inline_assembly* for the original ISA. This means that the number of instructions executed in the interpolation procedure is the best we can found. Since instruction $CHKB$ is independent of $LDMVBF$ and $BFLOAD$, we compare these results in three columns: CHKB_only, BF_only, and CHKBwithBF. Decoding 128 frames (one I-frame and followed by 127 P-frame) is used as our test data. The result is shown in Table. 4.1 sorted by original instruction count. We list the instruction count and the improvement compared to original code. By the experimental result, we can see that CHKB instruction can reduce the instruction count to 70.54% in average and BF instructions (LOADBF and LDMVBF) can reduce it to 71.31% in average. The instruction count will be decreased to 55.84% with both CHKB and BF instructions.

Table 4.1: Instruction count comparison

testbench	original	CHKB_only/Ratio	BF_only/Ratio	CHKBwithBF/Ratio
bridge	90026541	63492865 / 70.53%	79489699 / 88.30%	59088269 / 65.63%
salesman	126645892	90189827 / 71.21%	99377635 / 78.47%	75994915 / 60.01%
grandma	131720565	93895943 / 71.28%	101819945 / 77.30%	78102895 / 59.29%
news	147367027	104422772 / 70.86%	113643750 / 77.12%	87144116 / 60.61%
silent	169693030	119416696 / 70.37%	123078053 / 72.53%	95589033 / 56.33%
highway	209906220	147075994 / 70.08%	143759855 / 68.49%	112822003 / 53.75%
mthr_dotr	218442651	154154065 / 70.57%	149983980 / 68.66%	118099314 / 54.06%
carphone	375794438	265596751 / 70.68%	243518000 / 64.80%	194866382 / 51.85%
foreman	414850120	290957149 / 70.14%	262237307 / 63.21%	211057535 / 50.88%
tempete	461245930	326033782 / 70.69%	288495000 / 62.55%	234942516 / 50.94%
mobil	483655521	336245242 / 69.52%	304679189 / 63.00%	246154891 / 50.89%
average		70.54%	71.31%	55.84%

4.2 Hardware Overhead

Implementation of Fig. 3.3 and Fig. 3.4 are written in Verilog and are verified with Starfish. Input pattern that are generated randomly are fed to our design and DAG/DA design of Starfish to check the correctness of our design.

We use Design Compiler with UMC .18 library to synthesize our design and the original Starfish. Table. 4.2 shows the result of gate count and timing delay of DAG stage and Table 4.3 shows the result of DA stage.

Table 4.2: Gate count and timing delay of DAG stage

	DAG	DAG+	ours
gate count	3076	3513	3471
timing delay (ns)	5.99	5.99	5.99

Table 4.3: Gate count and timing delay of DA stage

	DA	DA+	ours
gate count	162	394	401
timing delay (ns)	4.09	4.21	4.25

We synthesize the Starfish code written by NCTU team and report the data in the column named 'DAG' in Table. 4.2 and the column named 'DA' in Table. 4.3. Then, we modify the Starfish code with our approach then synthesize it. The result is proposed in the column named 'DAG+' in Table. 4.2 and the column named 'DA+' in Table. 4.3. Finally, the column named 'ours' in Table. 4.2 and Table. 4.3 lists the result of rewriting the DAG and DA stage.