Chapter 3

Architecture and Software

In this chapter, we will present the architecture of Starfish designed by hardware team at NCTU. We implemented the design that supports $CHKB, BFLOAD$,and $LDMVBF$ instructions based on the Starfish architecture. To incorporate those new instructions, modifications to the tool-chain of Starfish will be introdued at the end of this chapter.

3.1 The Architecture of Starfish

3.1.1 Parallel Issue

Starfish, which is a Digital Signal Processor (DSP) that includes its own hardware core and tool-chain, has various length of instructions. In its ISA, most of the load/store instructions are encoded by 16 bits and arithmetic/vector/video instructions are encoded in 32 bits in length except some frequently used instructions. Starfish can issue up to three instructions in parallel and execute them in the same cycle. These three instructions must be one arithmetic/vector/video instruction along with two load/store instructions. Fig. 3.1 shows all the instruction types that Starfish supports and describes all kinds of instruction length.

3.1.2 Pipeline Issue

The architecture of Starfish contains seven pipeline stages which are IF, ID, OF, EX1/DAG, EX2/MEM, EX3/DA,and WB. Functions of each stage are briefly described in Table. 3.1.

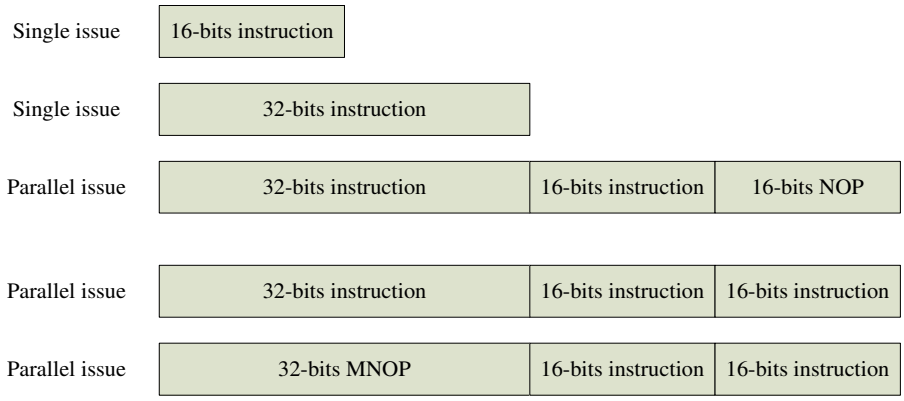

Figure 3.1: Single/parallel issue and three types of instruction length

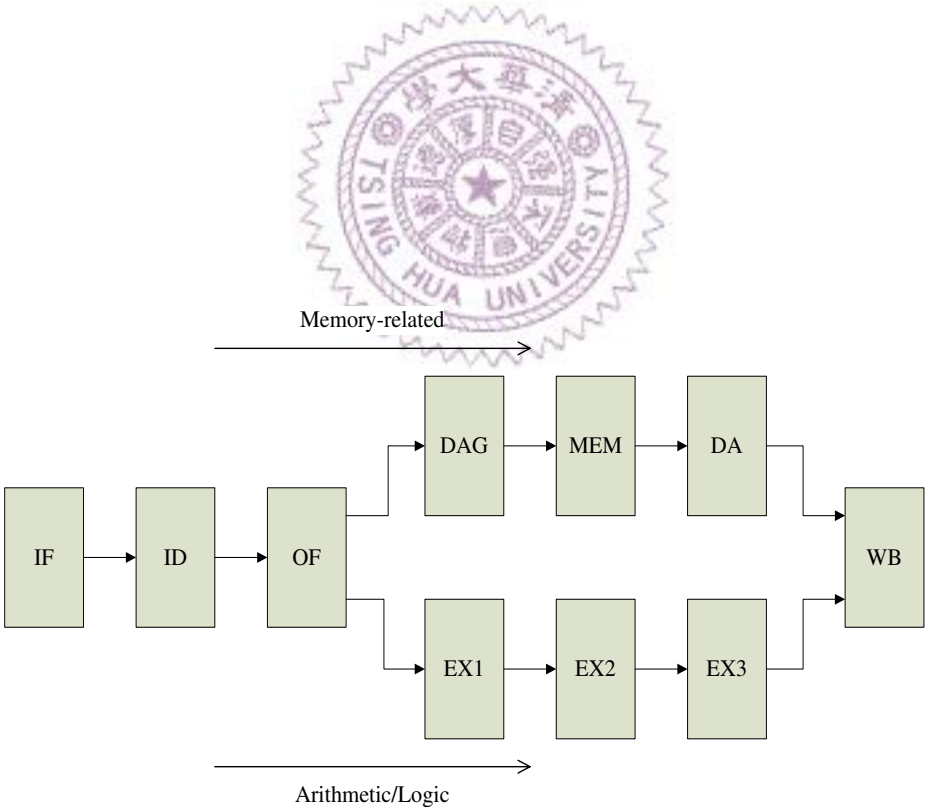

Figure 3.2: Pipeline stages of Starfish

Table 3.1: The descriptions of pipeline stages of Starfish

pipeline stages	briefly descriptions
IF	Instruction Fetch unit, which fetches instructions from instruction memory.
ID	Instruction Decode unit, which decodes control signal and operand number from instructions
OF	Operand Fetch unit, which fetches operand data from registers
EX1	Including Arithmetic/Logic/Vector/Video execution unit. Step1 of multiply operations
EX2	Step2 of multiply operations
EX3	reserved
DAG	Data Address Generator, which computes the address of Load/Store instructions.
MEM	Memory access of Load/Store instructions.
DA	Data Alignment, which extends the result by the option of instruction.
WB	Write back the result of computation into registers.

Starfish has two data paths of pipeline, one of which is for Memory-related operations, while the other is for Arithmetic/Logic operations. A graphical representation of data path is shown in Fig. 3.2. There are three types of execution unit in Arithmetic/Logic operation path. One is a Multi-Function Unit (MFU) in the EX1 stage that supports 32-bits arithmetic and dual 16-bits vector operations. Another is a video unit that supports quad 8-bits operation in the EX1 stage. The other is a multiplier accumulator in the EX1 and EX2 stages. The function units in Memory-related path includes DAG, MEM, and DA. The DAG stage (Data Address Generator) has two DAG units which can compute addresses in parallel. The MEM stage is the memory stage, and is to load the data from the address generated by DAG. The Data Alignment stage (DA) is used to perform extensions, such as zero-extension option.

3.2 Data Path for New Instructions

3.2.1 Hardware Design for CHKB instruction

We modify the original Memory-related data path for the new instructions. At first, we show the modification to the DAG design in Fig. 3.3. The modules in grey color are new modules we proposed, while other circuits are the original designs. The DAG has four data

input from registers, which are $src0, src1, src2$,and $src3$. The BR module outputs the bit-reversed data of the input. It is chosen only if the bit-reverse option is activated. The ADD module is a pure 32-bits full-adder, used in addressing circular buffering. The ADD/SUB module performs addition or substraction selected by the control signal *addorsub*. The CHKB module outputs the value of CHKB instruction, and finally selected by *If_chkb* which decide the value to be propagated to the MEM stage. *DAGoutput* will be drived by $CHKB$ module if If_chkb is 1. On the opposite, *DAGoutput* is drived by $OutMEM$ if If_chkb is 0.

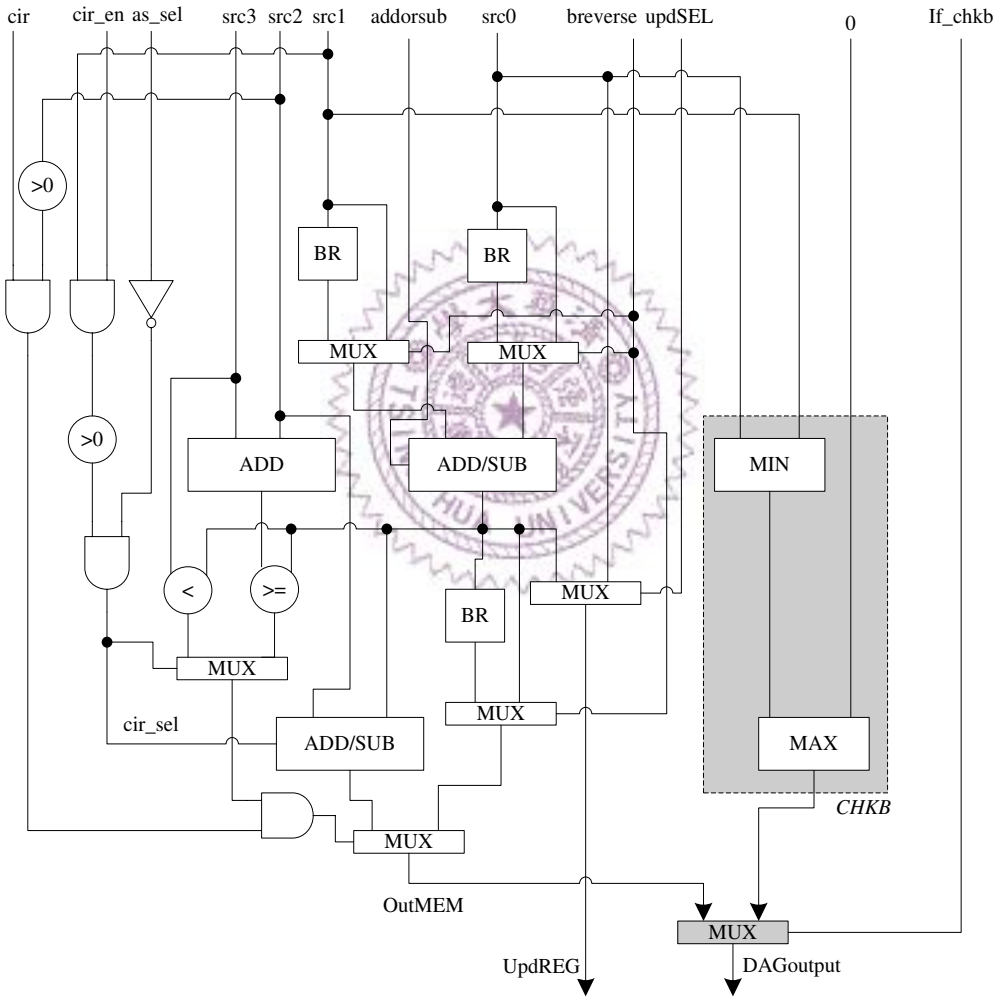

Figure 3.3: The block diagram of DAG that supports $CHKB$ instruction

3.2.2 Hardware Design for BFLOAD and LDMVBF Instructions

The second proposal to decrease the instruction count of interpolation procedure is a special buffer. The first issue is to locate the buffer. The buffer could be implemented at DAG stage to store the address calculated by previous iterations. Since that all the data loaded while interpolating are byte in size, we will have four times of overhead in buffer size if we store the address.

The second choice is the MEM stage. It is true that the buffer could work precisely when we place it at this stage. As the matter of timing issue, the MEM stage usually determines the timing of the whole machine. That is, if we implement the buffer at MEM stage, timing issue will be violated.

We now have the last chance of DA stage, however, this is the best choice by the result of experiment. The DA stage has much less circuits than all other stages (except EX3) because that the data-extension unit and data-alignment unit can be implemented easily. Although this stage has only a few circuits, it remains necessarily to be established to support the original ISA of Starfish. Fig. 3.4 shows the block diagram of DA stage with a new buffer.

Both kinds of buffer operation result in shifting the buffer by 8-bits. Difference between them is the data that stores into BF[4]. The control signal *BFwrite* will be set if the LDMVBF instruction is performing while the *BFread* control signal is activated by the BFLOAD instruction.

We design the buffer to have the ability of self-rotating by analyzing the property of the interpolation procedure and reducing the behavior of the new buffer. The procedure will *load* data for six times, and five of which are overlap as discussed in Chapter 2. As shown in Fig. 3.5, we can see that *BFread* is used here for rotating the data of the new buffer. It will decide which one of BF[0] and BF[4] is to be stored into BF[4]. If BFLOAD is performed, the buffer will be *rotated* and BF[0] will be stored into BF[4]. Meanwhile the original data of BF[4] will be propagated to BF[3] and so on. The programming of BFLOAD is written for five times so that the context of the new buffer will remain unchanged.

BFwrite is used to store the data into BF[4] propagated from lower order byte of data of MEM stage or from the buffer. When LDMVBF is performed, the buffer will be updated

with inserting a new data at the bottom. Whenever LDMVBF or BFLOAD instruction is performed, the data of {BF[1], BF[2], BF[3], BF[4]} will be moved to {BF[0], BF[1], BF[2], BF[3]}.

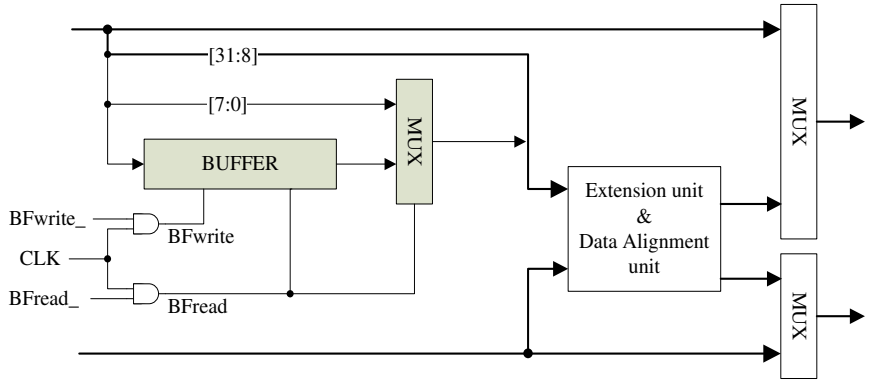

Figure 3.4: The block diagram of DA stage with the new buffer

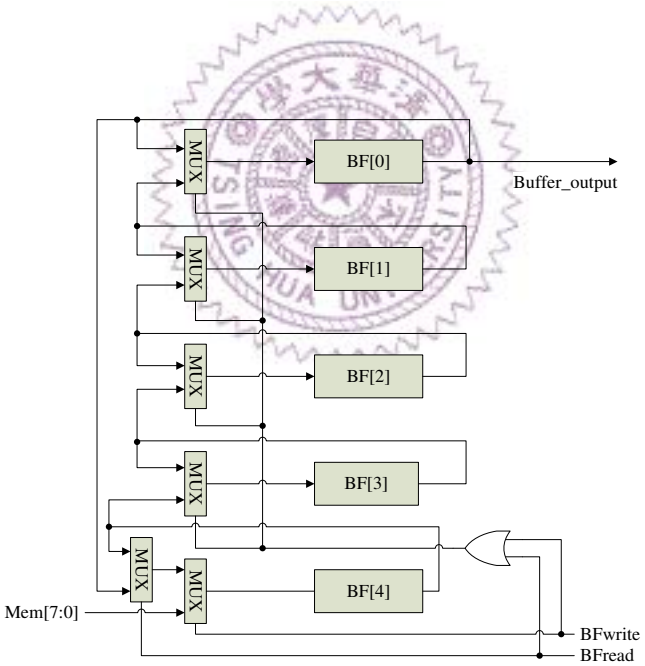

Figure 3.5: The block diagram of the new buffer

3.3 Encoding of New Instructions

The instruction format of the new instructions is described as follows. We encode our instructions as 16-bits instruction. In the encoding of ISA of Starfish, the leading two bits

decide the instruction types: '00' and '01' are the field of 16-bits instructions without parallel issue, '10' is the field of 16-bits instructions with parallel issue, and '11' is the field of 32-bits instructions. The CHKB instruction is not required to be executed in parallel and is encoded leading by '01'. Load/Store instructions that can be executed in parallel are encoded '10' at the most significant bits. We encode LOADBF and LDMVBF in the '10' field since they are designed with parallel issue. Fig. 3.6 shows the encoding of $CHKB$ instruction and Fig. 3.7 shows that of $BFLOAD$ and $LDMVBF$ instructions.

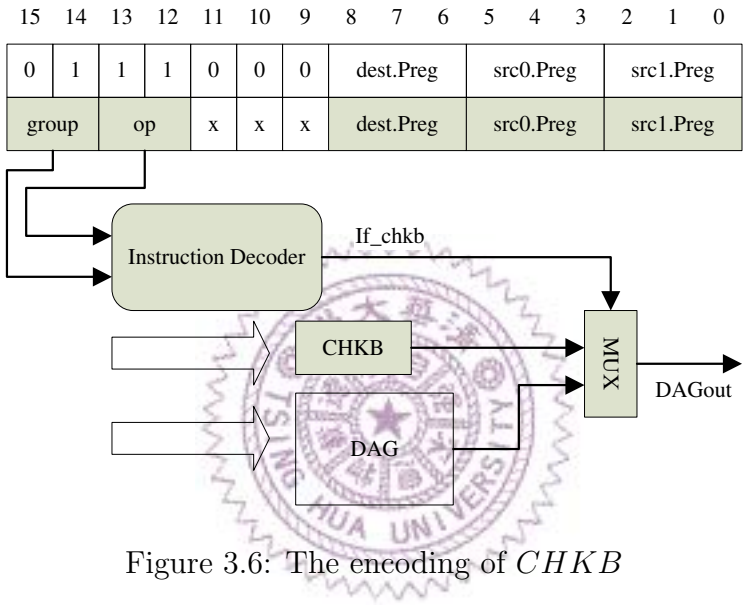

Figure 3.6: The encoding of $CHKB$

3.4 Tool-Chain of Starfish

After designing the data path, we have to develop software to support our new instructions. Partial source code of Starfish tool-chain, including the assembler and the simulator, is modified to recognize our new instructions. We add the assembly syntax and the corresponding binary code for our new instructions to Starfish-Assembler and also add the behavior (including the new buffer) of new instructions to Starfish-Simulator.

We now demonstrate how to modify Starfish-Assembler. The first task before adding the new instruction into Starfish-Assembler, $LDMVBF$ for example, is to modify the token parser by adding the keyword $LDMVBF$ so that the token parser takes the keyword as a

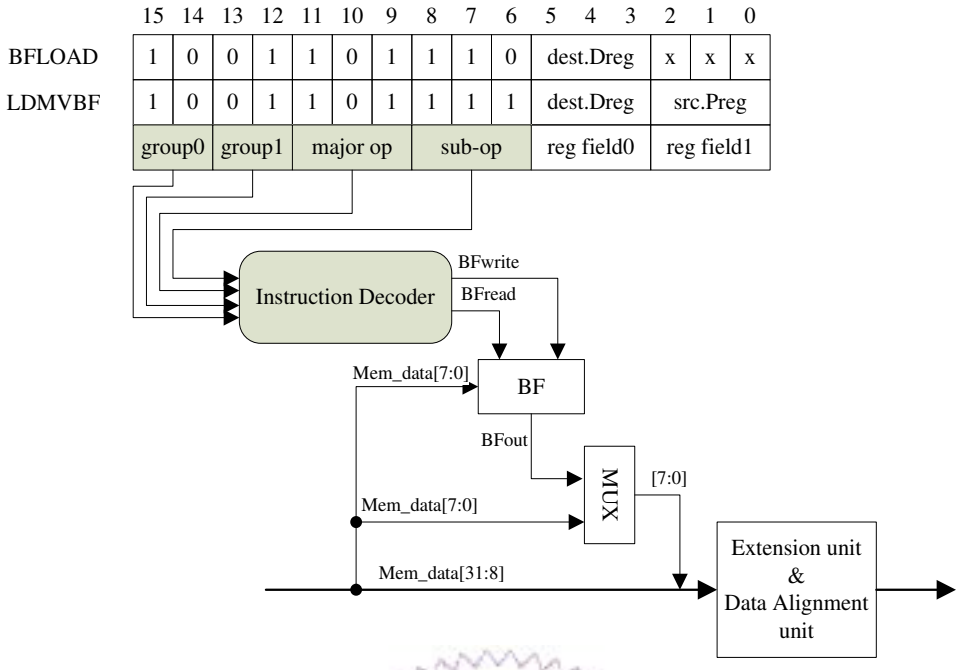

Figure 3.7: The encoding of $BFLOAD$ and $LDMVBF$

legal token. Then the token parser transmits the token sequence ($LDMVBF$) to the syntax parser of assembler. We can add the syntax rule for our new instructions after all these steps were done. The syntax rule contains the field information including the length and position of op-code, source field, and destination field. According to the new rule, the syntax parser will then generate the corresponding binary code of this new instruction. Modification flow of the assembler is shown in Fig. 3.8.

Starfish-Simulator is to be modified in the second step. New behavior files of the corresponding binary code are placed in the simulator. Those files include the code range, operand field, and operation of the new instruction. To built up the buffer, the simulator needs a file that describes the actions performing on the buffer. Like the file that describes the memory, the file of buffer must be included in the the simulator core engine. Fig. 3.9 shows the modification of simulator.

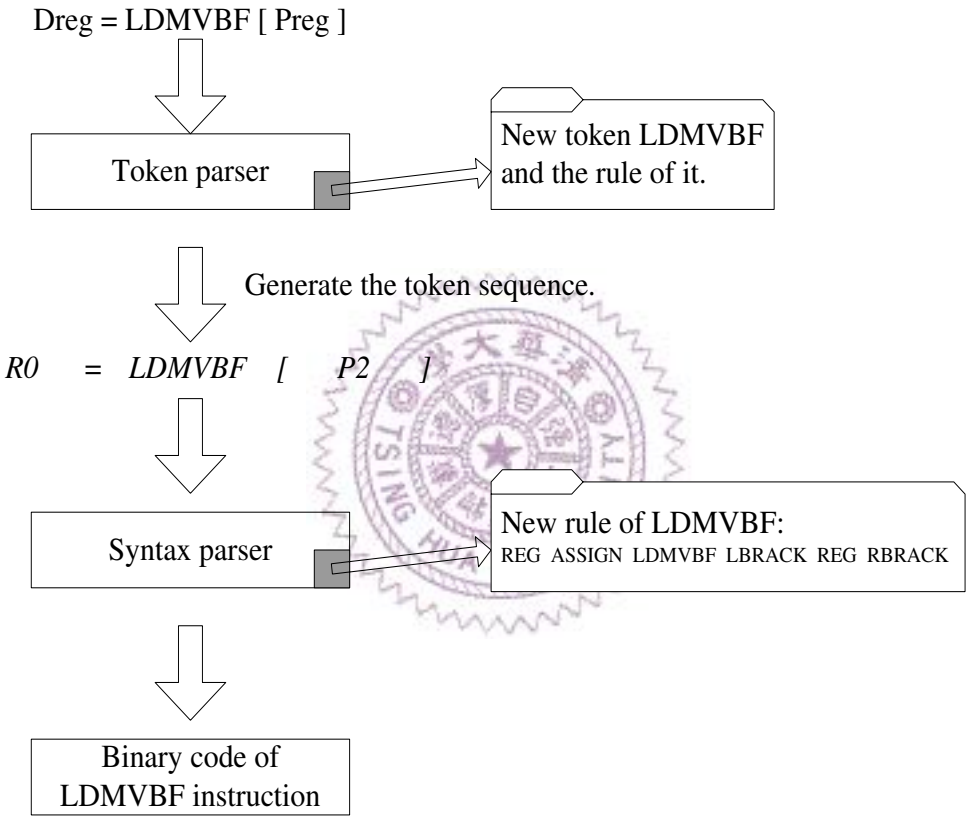

Figure 3.8: Modification to assembler

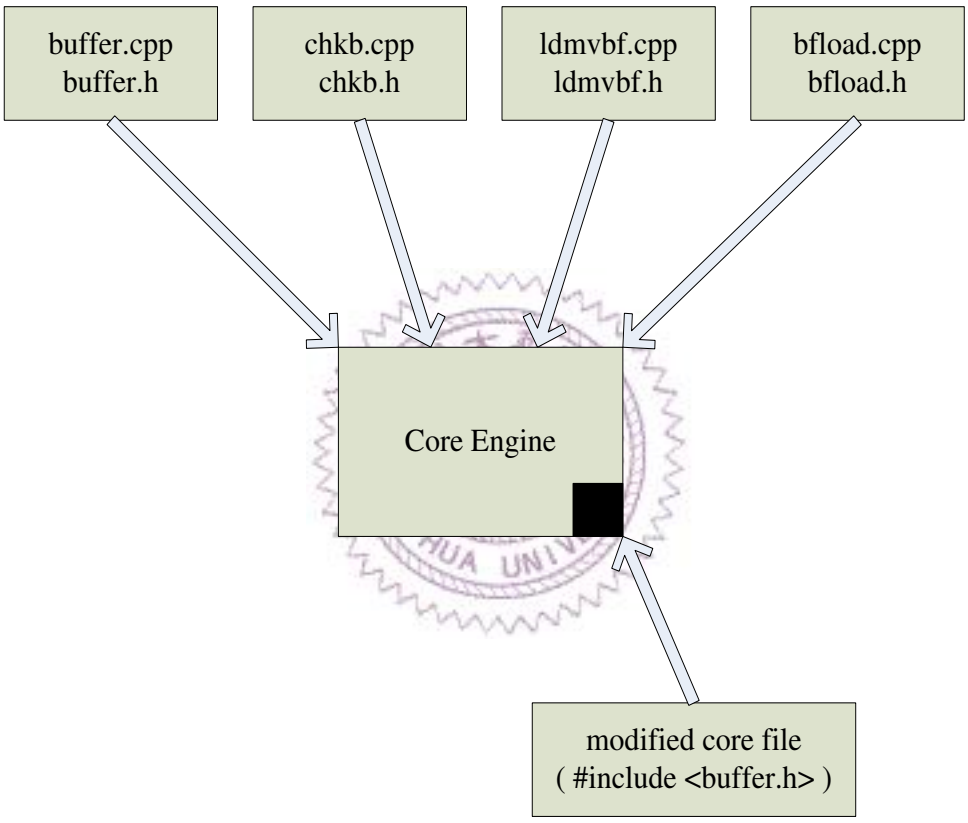

Figure 3.9: Modification to simulator