Back-end porting of FT_MX based on LLVM compilation architecture

Ping Deng[1], *Xiaolong* Zhu[1], *Haiyan* Sun[1,*], and *Yi* Ren[1]

[1]College of Computer, National University of Defence Technology, 410073 Changsha, China

Abstract. The processor FT_MX is a high-performance chip independently developed by the National University of Defense Technology, with an innovative architecture and instruction set. LLVM architecture is a widely used and efficient open source compiler framework initiated by the University of Illinois. This paper introduces the basic architecture and functions of LLVM, analyzes the back-end migration mechanism of the architecture in detail, and gives the specific process of implementing FT_MX back-end migration, and realizes the support of LLVM architecture to the back-end of FT_MX processor.

1 Introduction

With the widespread use of integrated circuits and the rapid development of microprocessors, the performance requirements for system power consumption are getting higher and higher. As users pursue the goal of smaller chip size, lower power consumption and faster speed, manufacturers also need to take into account factors such as rapid production and profit maximization, which brings greater challenges to the design work of engineers. Compilers, which are between modern programming languages and hardware structures, can deal with most of the above-mentioned challenges by optimizing program performance. But for the hardware resources of embedded systems, the diversity of embedded systems makes different system platforms correspond to different software and hardware development platforms, so when a new microprocessor is used, a new compiler is needed to support it.

It is an arduous task to develop its compiler from scratch corresponding to a specific architecture. At present, a method based on open source code and a transplantation strategy is used to construct the compiler. This method provides a complete set of interfaces for defining the hardware architecture, and the interfaces provided can support the existing mainstream processor structure. Currently commonly used open source codes mainly include GCC, LLVM, SUIF, LCC, Open64, ORC, etc. This article mainly discusses building the FT-MX compilation system within the LLVM compilation framework.

* Corresponding author: helen@nudt.edu.cn

2 LLVM compilation architecture and FT_MX DSP analysis

2.1 LLVM basic architecture analysis

The LLVM project is a project initiated by the University of Illinois. It is a widely used and efficient open source compiler framework. The project reuses GCC's front-end high-level language processing, while extracting the characteristics of high-level languages such as C and C++, and adopts a self-created code optimization mechanism to make the global optimization of the entire program possible[1]. From the perspective of the LLVM compilation process, it uses a three-stage design similar to traditional static compilers (C compilers such as GCC, Free Pascal, etc.), which consists of a Clang front-end, an optimizer and a back-end. Among them, Clang is a highly modular and lightweight compiler, which has the advantages of fast compilation speed, small memory footprint and convenient secondary development. The main function of the front-end is to parse the program source code, perform lexical and grammatical analysis, and generate an AST (Abstract Syntax Tree) based on language characteristics, and further convert the AST into an intermediate representation of a specific grammatical format. The optimizer receives the intermediate representation output by the front-end and completes a series of optimizations, the back-end translates the intermediate representation into a language that can be recognized by the machine according to the target hardware information provided. Figure 1 shows the architecture under the basic compilation process of LLVM. From the basic functional composition of LLVM source code, the LLVM architecture can also be divided into three parts[2]: LLVM virtual instruction set, integrated library for analysis, optimization, code generation, etc., and tools based on the integrated library[3-4] such as assembler, linker, debugger, etc. Figure 2 shows the structure of the LLVM architecture divided by basic functions.

Compared with traditional compilers, LLVM has the following advantages: the intermediate representation suitable for optimized operations and independent of the platform, namely IR (intermediate representation), a highly modular code structure that is convenient for back-end addition, a clear pass process and support writing application-specific optimized passes. Its unique modular design and unified IR make it suitable for GPU databases, deep learning inference frameworks, blockchains and many other fields. For example, LLVM serves as a bridge connecting the underlying hardware and software framework in a deep learning[5] compiler (such as TVM), which can solve the incompatibility problem when different upper-layer applications are deployed on different hardware targets.

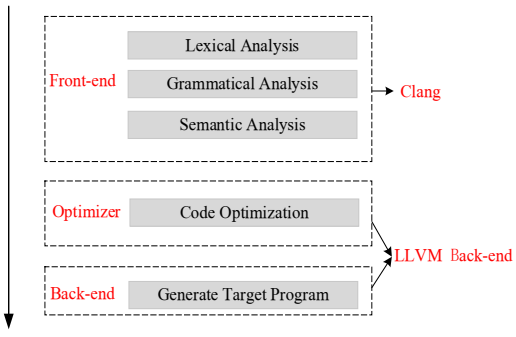

Fig.1. The LLVM architecture under the compilation process.

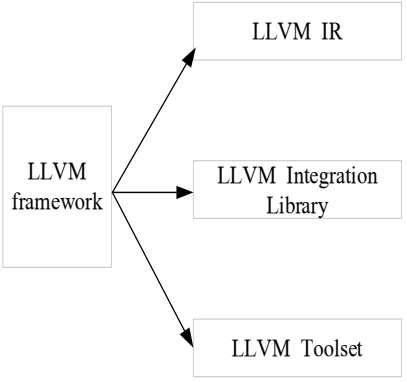

Fig.2. The LLVM architecture under the basic functions.

2.1.1 LLVM virtual instruction set

The LLVM virtual instruction set (which also called LLVM IR) is an instruction set similar to RISC (Reduced Instruction Set Computer), it uses an intermediate representation in the form of SSA (Static Single Assignment)[6]. The main features of the intermediate representation are three-address instructions and unlimited number of registers, and because the representation has a syntax independent of high-level language and target hardware, code analysis and optimization are easy. In fact, LLVM IR is defined in three isomorphic forms[7]: instruction and CPP-like memory representation, serialized binary disk representation (*.bc file), and human-readable text format disk representation (*.ll file). And there are corresponding tools to support the mutual conversion of *.ll files and *.bc files, llvm-as supports the conversion from *.ll files to *.bc files, and llvm-dis is the opposite.

2.1.2 LLVM integrated library

LLVM's integrated library contains multiple sub-libraries[8], such as a core library for parsing and processing intermediate code, an analysis library that provides control analysis and data analysis functions, a conversion library for optimizing intermediate code conversion, a code library for generating target code, the target processor library of the back-end information description function, and the runtime library that supports interpretation and execution. All in all, the LLVM integrated library provides most of the functional support in the compiler development process, including classes used to describe basic blocks, functions and modules, pass definition and pass management functions in compilation optimization, and data information analysis in the compilation process , basic debugging functions and various optimization processing. Figure 3 shows the basic composition of the integrated library and its corresponding functions.

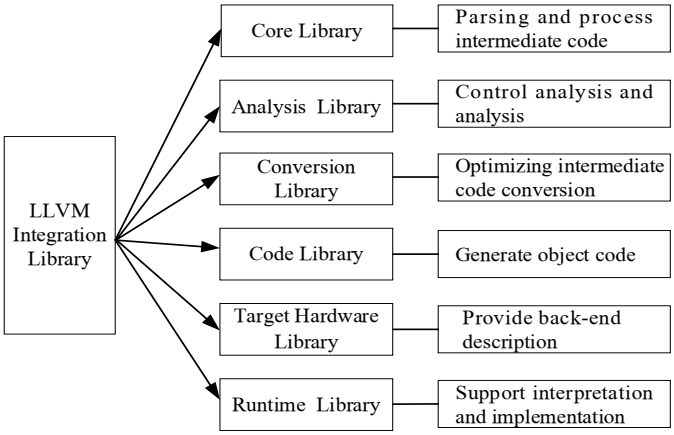

Fig. 3. The basic composition of the integrated library and its corresponding functions.

2.1.3 LLVM toolset

In order to better carry out the follow-up development work, and make the program can be optimized in the five stages of compiling, linking, installation, running, idle time, etc., a large number of practical tools must be provided to build the necessary Operating environment. The tools provided by LLVM can be divided into the following categories: compilation tools, debugging tools, back-end tools, and basic tools. Table 1 shows the tools provided by llvm

and their basic functions in the compilation process.

Table 1. Tools provided by llvm and their basic functions in the compilation process.

category	name	Features
Compilation tool	opt	Optimize bytecode with LLVM
Debugging tools	bugpoint	Locate LLVM tools or compile errors
	llvm-extract	Strip the specified function from a large program
	llvm-bcanalyzer	Check the encoding of the .bc file
Backend tools	llc	Generate this machine code for a byte code file
	lli	The .bc code of the target processor can be directly run locally
	tblgen	Transform the description of the target processor into the corresponding description source code file, thereby simplifying the migration of the back-end target processor
Basic tools	llvm-as	Assemble human-readable text files into byte codes.
	llvm-dis	Transform byte code files into human-readable files
	llvm-link	Concatenate several byte code files into one file
	llvm-nm	Print name and symbol type in bytecode file
	llvm-ar	Bytecode file
	llvm-ranlib	Index the files generated by llvm-ar

2.2 FT_MX processor

The processor FT_MX is a DSP chip independently developed by the National University of Defense Technology. It adopts the 11-transmitted VLIW (Very Long Instruction Word) structure (scalar 5 outflow, vector 6 outflow) and a single instruction with a width of 40 or 80 bits. The processor has functional components such as instruction fetching unit, vector processing unit VPU (Vector Process Units), scalar processing unit SPU (Scalar Process Units), vector storage unit AM (Array Memory) and DMA (Direct Memory Access), and its supporting FT_MX instruction set provides corresponding scalar instructions and vector instructions. Due to FT_MX's unique architecture and independent innovation instruction set, the existing compilers cannot meet its application requirements,only compilers specific to the chip can be developed. In order to make FT_MX have a broader application platform and service range, a compilation system for FT_MX will be built under the LLVM framework. Figure 4 shows the basic architecture of FT_MX.

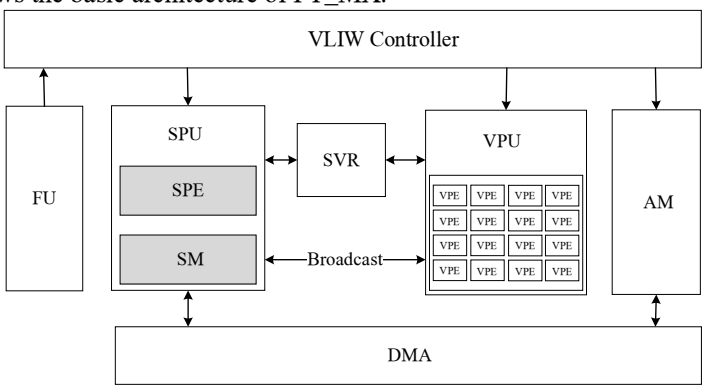

Fig. 4. The basic architecture of FT_MX.

3 Implementation of FT_MX backend porting

3.1 FT_MX global implementation description

To make the LLVM upper layer structure obtain the information of the FT_MX architecture, the global description files MXTargetMachine.h and MXTargetMachine.cpp need to be added, and functions such as getFrameInfo( ), getRegisterInfo( ), getInstrInfo( ) and getTargetData( ) are written according to the hardware corresponding information. At the same time, use the following code to indicate the byte order, alignment and type size information of FT_MX. And in order to make LLVM instructions can be selected and printed correctly,need to add the corresponding addInstSelector( ) and addAssemblyEmitter( ) functions.

3.2 FT_MX register implementation description

In order to realize the basic description of register information, it is necessary to realize the description of the following types of registers in MXRegisterInfo.td: 16 base address registers, 16 address offset registers, 64 general registers, and 64 vector registers. The register definition gives the name, number and register alias information of each register, and classifies the above-mentioned registers according to the register usage conventions corresponding to the instruction set, and establishes GPRRegs, FPRRegs, VPRRegs, APRRegs, VAPRRegs, OPRRegs, VOPRRegs and other register grouping. Among them, GPRRegs is a scalar general register group (including R0~R63), and specifies R10, R12, R14, R16, R18, R20, R22, R24 as parameter transfer registers, in particular, R10 as the return value register and R63 as the return address register , R0~R6 are used as scalar condition registers. FPRRegs is a floating-point register group (including R0~R63), and the specific register function division is similar to GPRRegs. VPRRegs is a vector register group (including VR0~VR63), and VR0~VR6 are vector condition registers. APRRegs is a scalar base address register group (including AR8~AR15), and VAPRRegs is a vector base address register group (including AR0~AR7), where AR8 and AR9 are respectively the frame register and stack register. OPRRegs is the scalar offset register group (including OR8~OR15), and VOPRRegs is the vector offset register group (including OR0~OR7).

3.3 FT_MX instruction set description

The instruction set description needs to complete a clear description of the FT_MX instruction format, instruction operands, addressing mode, instruction function and encoding, assembly code output, and the matching relationship between instruction and LLVM IR. According to the characteristics of the FT_MX instruction set, the base class MXInst is written through the Instruction class provided by LLVM and inherited from the MXInst class. 16 types instruction formats are defined in MXInstrFormats.td, and corresponding field rules are defined for each format as an instruction template. In MXInstrInfo.td, according to the classification definition instruction defined in MXInstrFormats.td, it indicates the operand code value of the instruction, the type of assembly output format, and the matching relationship with LLVM IR and special matching requirements.

3.3.1 Arithmetic operation instructions

The implementation of arithmetic operation instructions is relatively simple. After completing the inheritance of the base instruction template, add the correct matching

description to the LLVM IR and the corresponding instruction. Take the addition operation instruction as the representative of the arithmetic operation instruction, and complete the process description of adding the arithmetic operation instruction. In the FT_MX compiler, addition is divided into four categories, namely 32-bit signed addition, 32-bit unsigned addition, 64-bit signed addition, and 64-bit unsigned addition. For each type of addition, the FT_MX compiler is divided into register and register addition, register and immediate value addition. The following shows the instruction definitions for 32-bit signed and unsigned addition.

```
def   SADD32I:1OP_2OP_Sirr<0b0100101100,IIC_ADD_SUB_I,"sadd32",unit_m_s>;
def   SADD32R:1OP_2OP_rrr<0b0100111100,IIC_ADD_SUB_R,"sadd32",unit_m_s>;
def   SADDU32I:1OP_2OP_Uirr<0b0100100100,IIC_ADD_SUB_I,"saddu32",unit_m_ s>;
def   SADDU32R:1OP_2OP_rrr<0b0100111100,IIC_ADD_SUB_R,"saddu32", unit_m _s>;
```

3.3.2 Logic Operation Instructions

The logic operation instructions of FT-MX include AND, OR, XOR, NOT, and similar to the addition of arithmetic operation instructions, we need to specify the basic instruction format and then correspond to LLVM IR to complete the matching instructions. The AND operation instructions are listed below as representative.

```
def   SANDI:1OP_2OP_Sirr<0b1111011000, IIC_SIALU_AND_I, "sand",unit_l_s>;
def   SANDR:1OP_2OP_rrr<0b1110011000, IIC_SIALU_AND_R, "sand",unit_l_s>;
```

3.3.3 load/store instruction

FT_MX architecture uses two addressing modes: register relative addressing and base address plus index addressing. In MXInstrInfo.td, the two addressing modes are modeled as Store_RIR, Store_RRR and LoadI, LoadR respectively. The following is represented by the Store command.

```
Class Store_RIR<bits<4>funct4,string opcodestr>:MXInstMI<funct4,(outs),(ins GPR: $dst,
mem_op_s:$ptr,uimm10:$uimm10),opcodestr, "$dst, $ptr " ,unit_ls_s >
      {let Itnerary=IIC_SLS_STORE_I;}
Class Store_RRR<bits<4>funct4,string opcodestr>:MXInstMR<funct4,(outs),(ins GPR: $dst,
mem_op_s:$ptr,OPR:$offsetR),opcodestr, "$dst, $ptr " ,unit_ls_s >
      {let Itnerary=IIC_SLS_STORE_R;}
```

Two addressing modes, define a memory operand using addressing mode.

```
def mem_op_s: Operand<XLenVT>{
      let PrintMethod = "printMemOperand";
      let MIOperandInfo=(ops APR);}
```

In the implementation of the load/store instruction set, the addressing mode and memory operand defined above need to be used, such as:

```
def   SSTW_I: Store_RIR<0b0101, "sstw">;
def   SSTW_R: Store_RRR<0b0110, "sstw">;
```

3.4 Implementation of FT_MX instruction selection

The instruction selection description of FT_MX includes two aspects: the legalization process and the instruction matching process. Legalization is mainly achieved through the files MXISelLowering.h and MXSelLowering.cpp. LLVM uses the SelectionDAG structure related to the underlying data to represent IR for instruction selection, and calls the instructions or data not supported by the target platform in SelectionDAG as illegal instructions or illegal data Type, an illegal DAG containing illegal instructions or illegal data types, needs to go through the steps of DAG legalization to be converted into a legal DAG

supported by the target platform. DAG legalization includes data legalization and instruction legalization. Instruction legalization is to convert illegal DAG instructions into instructions supported by the target platform. Custom is usually used to complete the legalization of instructions. Data legalization converts unsupported data types. Generally, the legalization of data types is completed by upgrading small data types to large data types, that is, the Promote operation, or converting large data types to small data types, that is, the Expand operation. If the target architecture only supports 32-bit integer data types, the 8-bit or 16-bit integer data types in the DAG must be promoted to 32-bit integer data types. For large data types such as 64-bit integers, it needs to be extended to two 32-bit integer data types.

Table 2. Tests and Results.

Test case 1: Integer and floating point addition and subtraction
int main()
{
int a,b,c;
float A,B,C;
a=3;
b=4;
c=a+b;
A=5.0;
B=2.5;
C=A-B;
printf("%d,%f",c,C);
}
	Assembly output result 1:
	……
	smovi24 3, r50
	ssth r50,*-ar8[12]
	snop 3
	smovi24 4, r50
	ssth r50,*-ar8[16]
	snop 3
	sadd.M1 r50,r51, r50
	smovi 1084227584, r50
	ssth r50,*-ar8[24]
	snop 3
	smovi 1075838976, r50
	ssth r50,*-ar8[28]
	snop 3
	sfsubs32.M1 r50,r51, r50
	……

Test case 2: for loop
int main()
{
int a=1;
int i;
for(i=0;i<5;i++)
a+=a;
printf(“%d”,a);
}
	Assembly output result 2:
	……
	.LBB0_1:
	slt r50,r51,r50
	smov.M1 r50, r0
	[!r0] sbr .LBB0_4
	snop 6
	sbr .LBB0_2
	snop 6
	.LBB0_2:
	sadd.M1 r50,r50, r50
	ssth r50,*-ar8[16]
	snop 3
	sbr .LBB0_3
	snop 6
	.LBB0_3:
	sadd.M1 1,r50, r50
	ssth r50,*-ar8[20]
	snop 3
	sbr .LBB0_1
	snop 6
	.LBB0_4:
	sldh *-ar8[12],r10
	snop 6
	……

The instruction matching process is mainly implemented in the file MXISelDAGTo-DAG.cpp, in which the function Select( ) is overloaded, and the call to SelectCode ( ) in

MXGenDAGISel.inc is added in the function body. The function SelectCode( ) is generated by the MXInstrInfo.td file. The description of the instruction pattern matching in MXInstrInfo.td will produce the method of instruction selection. Sometimes it is necessary to manually write code in Select( ) to handle instructions that cannot be selected using SelectCode( ) , In order to achieve its matching with LLVM IR.

3.5 Implementation of FT_MX assembler exporter

The assembly output format of the instruction is defined in MXInstrInfo.td. For example, the assembly output format of the instruction opcodestr is opcodestr, "$imm6,$src2,$dst". MXInstrInfo.td is compiled by TableGen to generate the assembly output function printInstruction( ) located in MXGenAsmWriter.inc. The function runOnMachineFunction( ) in MXAsmPrinter.cpp calls this function to realize the print output of instructions.

3.6 FT_MX compilation system support implementation

In order for the LLVM compilation architecture to support the back-end description of FT_MX, you need to add the file MXAsmPrinter.cpp under the lib/Target/MX/Asmprinter directory, and add *.h, *.cpp and *.td under the lib/Target/MX directory. Create a makefile in the lib/Target/MX directory, and specify the target name TARGET, compilation level LEVEL, and library file name LIBRARYNAME in the file, and include the Makefile.common file in the top directory , to achieve support for the makefile system. Then modify the configure script file in the top directory and add the FT_MX target in the TARGETS_TO_BUILD variable. Re-run the configuration script file and compile the entire LLVM project, and we will get the LLVM that supports the FT_MX target platform.

4 Overall system test

In order to verify the completeness of the instruction set and the correctness of the back-end support in the FT_MX back-end migration process, the integer and floating-point addition and subtraction examples are selected to illustrate the support for basic arithmetic instructions, and for loop examples to illustrate the support for judgment, loop and other functions. Table 2 shows the test cases and the corresponding output results.

According to the above test results, it can be seen that according to the basic process of FT_MX back-end migration in Section 3, after adding arithmetic operation instructions, load/store instructions, logical operation instructions and other instructions, the assembly code corresponding to the FT_MX instruction set can be correctly generated.

5 Summary

Based on the basic architecture and functions of LLVM, this paper analyzes the back-end migration mechanism of the architecture in detail, and gives the specific process of implementing FT_MX back-end migration. With the support of the back-end migration mechanism of the LLVM compilation system, the description of the various attributes of the FT_MX architecture is completed, the addition of FT_MX to the back-end of the LLVM compiler is realized, and the correct assembly code can be generated for FT_MX.

References

1. Dong Feng,Fu Yuzhuo. ARM back-end porting based on LLVM architecture[J].

Information Technology, 2007(07):38-41.

2. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. International Symposium on Code Generation and Optimization, Palo Alto, 2004:75~86.
3. Ebner Dietmar, Brandner Florian, Krall Andreas. Leveraging predicated execution for multimedia processing. 2007 5th Workshop on Embedded Systems for Real-Time Multimedia, Salzburg , 2007:85~90.
4. Bolat Murat, Li Xiaoming. Context-aware code optimization. 2009 IEEE 28th International Performance Computing and Communications Conference, Scottsdale , 2009:256~263.
5. Chen Zhaoyun , Quan Wei , Wen Mei , et al. Deep Learning Research and Development Platform: Characterizing and Scheduling with QoS Guarantees on GPU Clusters[J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 31(1):34-50.
6. Han Yongjie. Analysis of LLVM Compiler System Structure and ARCA3 Backend Porting[D]. Harbin Institute of Technology, 2010.
7. https://blog.csdn.net/night_zw/article/details/55100646.
8. Dong Feng. LLVM Compiler System Structure Analysis and Back-end Migration. Shanghai Jiaotong University Master's Thesis. 2007:9~14.