Design and Implementation of Assembler for High Performance Digital Signal Processor (DSP)

Peng Ding[1], Haoqi Ren[1], Zhifeng Zhang[1], Jun Wu[1](✉), Fusheng Zhu[2], and Wenru Zhang[2]

[1] Tongji University, No.4800 Caoan Road, Jiading District, Shanghai, People's Republic of China
657799191@qq.com, {renhaoqi,zhangzf,wujun}@tongji.edu.cn

[2] GuangDong Communications & Networks Institute, Guangzhou, Guangdong Province, China
{zhufusheng,zhangwenru}@gdcni.cn

Abstract. With the rapid development of the fifth-generation mobile communication technology (5G), existing digital signal processors (DSP) on the market cannot efficiently provide the performance required by some applications. In this situation, we design a new DSP with faster speed, lower latency and higher performance. In this article, based on the new DSP which can adapt to the new technology of 5G, we designed an assembler called Swift Assembler (SA). Different from the traditional assembler, SA is based on the Gnu Architecture Description Language, (GADL). We perform semantic analysis on GADL description files and then with the help of flex, bison and Binutils, the assembler is compiled and generated. With the support of GADL, SA has a clearer architecture and better scalability. At the same time, it covered the underlying implementation. Benefit from this, programmers can modify its source code with no need to understand the underlying implementation process. In this way, the design of interdependent hardware and software can be more easily.

Keywords: 5G · DSP · GADL · SA

1 Introduction

Recently, the fifth-generation communication technology (5G) [1] has become a hot topic. With the continuous advancement of communication technology worldwide, 5G has developed rapidly and matured. Theoretically, 5G can provide a peak data rate of 20 Gbps with ultra-low latency of 1 ms. In smart grid, monitoring system, asset tracking, connected cars and more, it has a great application value. In the next few years, the market of wireless communication belongs to 5G. However, the existing data signal processor (DSP) cannot efficiently provide the qualified speed, latency and overall performance required by the upgrading of 5G in the application of LTE-Advanced Pro and Gigabit wireless networks. Because of this, there is an urgent need for a new DSP that can meet the

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2020
Published by Springer Nature Switzerland AG 2020. All Rights Reserved
H. Gao et al. (Eds.): ChinaCom 2019, LNICST 312, pp. 741–751, 2020.
https://doi.org/10.1007/978-3-030-41114-5_56

requirements of ultra-high frequency and ultra-low power consumption, providing large-scale computing power, and supporting high-precession algorithms. We have done much work on the new DSP. In our design, the new DSP has a flexible architecture and a variety of optional features. Also, it implements new instructions for baseband processing, ultra-low transmission delay of the core and accelerator, and flexible customer provisioning and scaling to meet the needs of large-scale user management and multi-RAT (wireless access technology). At the same time with the research of hardware, our work on software have also made great progress. We have finished the work of toolchain based on the new DSP. In this article, the design of assembler will be introduced in detail.

The traditional assembler implementation usually builds a mapping of assembly code to machine code with the help of high-level programming languages. And someone may prefer to use existing assembly tools such as GNU Binutils to reduce the work of programming [2]. But int this way, it usually involves the underlying implementation in the design of the assembler. This will bring great difficulties to the use and modification of other programmers.

In our design of assembler, we used Gnu Architecture Description Language, (GADL) [3, 4] to help us complete the mapping of the assembly code to the machine code. In this way, we build a clear architecture of assembler, reduce the code we need to finish and make the whole project easy to read and modify for other programmers. We ported the Binutils toolset to help us to read in the assembly files and complete some relocation work. But for others, they don't need to understand how we do this work. They can use it and modify it for their own requirement through the upper layer codes.

2 Description of the New DSP

This new DSP is aimed to achieve high performance and low power consumption and its main application field is wireless communication. In addition to the most advantages of traditional DSP, it also has autonomously controllable high-performance instruction set architecture (ISA), dedicated acceleration instructions and high-performance DSP core microarchitecture for independent intellectual property rights and more.

This DSP support high parallelism single instruction multiple data (SIMD), providing a 640-bit SIMD unit that can perform vector operations of 16-channel-40-bit, 32-channel-20-bit and 64-channel-10-bit. At the same time, it can emit 8 instructions simultaneously in VLIW. To achieve this, we use a 10-stage pipeline as shown in Fig. 1.

Fig. 1. A 10-stage pipeline

In the structure, the DSP includes a DSP core, a direct memory access (DMA) module, on-chip memory (I-MEM, D-MEM) and a debug module as the Fig. 2 shows.

The DSP core uses the LOAD/STORE instruction to directly access the data in the on-chip memory or to transfer data to and from external device and memories via the bus. And the on-chip memory transfers data through the DMA channel. The debug module provides an interface to the external debugger.

3 Description of GADL

Swift Assembler (SA) is an assembler based on the GADL. GADL is a highly scalable architecture description language presented by one of our members in our team. It's usually applied to the design of toolchain. The structure of GADL is described in Fig. 3.

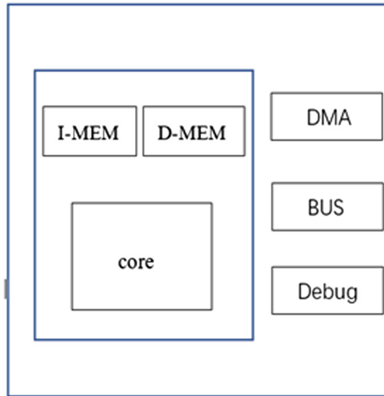

Fig. 2. A simplified structure of DSP

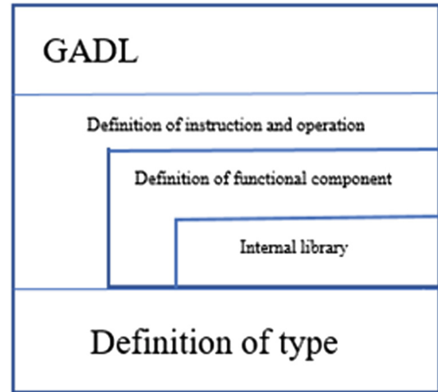

Fig. 3. Structure of GADL

Similar to lisp [5], the syntax of GADL uses a prefix expression. For example, the Backus-Naur Form (BNF) of the definition of an integer in the grammar is as follows.

- INT:: == '(' 'int_type' LIST ')'
- LIST:: = [LIST] DEC
- DEC:: == '(' NAME WIDTH FLAG ')'
- WIDTH:: = '(' 'width' NUMBER ')'
- FLAG:: == '(' 'flag'<signed | unsigned> ')'

An integer description begins with the keyword "int_type" followed by several definition lists (DEC). Each of the integer types begin with the type name followed by two attribute definitions: data width and data type (signed and unsigned). An example to help understanding is given below.

- (int_type
-     (bf5
-         (width 5)
-         (flag signed)
-     )
-)

In the example, we define a signed integer "bf5" and its width is 5-bits. Similarly, we can define address type with GADL as follows.

- ADDRESS:: == ‘(’ ‘address’ LIST ‘)’
- LIST:: == [LIST] DEF
- DEF:: == ‘(’ NAME WIDTH PCREL SHIFT ‘)’
- WIDTH:: == ‘(‘ ‘width’ NUMBER ‘)’
- PCRA:: == ‘(‘ ‘pcra’ < ‘TURE’ | ‘FALSE’> ‘)’
- SHIFT:: == ‘(’ < ‘right_shift’ | ‘left_shift’> NUMBER ‘)’

In the BNF, the definition of address begins with keyword ‘address’ followed by three attribute definitions: width, pcra, shift. The attribute pcra indicates whether this address is a PC relative address. And the attribute shift indicates how to shift the value of the address in the address resolution. An example is given below.

- (address
- (addr21rel
- (width 21)
- (pcra TURE)
- (right_shift 2)
-)
-)

With the help of these meaningful definitions, we can describe the instruction set completely. Based on the instruction set described, the design of the assembler is easily.

4 The Design of Assembler

The design of assembler usually contains two aspects. One is to transfer the instructions to the machine codes and the other is to identify assembly files. In the other word, just like compiler, the design of assembler can be divided into lexical analysis and semantic analysis [6, 7].

To implement the translation from instructions to machine codes, the most designs of traditional assemblers are to use lots of code to describe the mapping of instructions to machine codes [8]. In this way, the amount of code for the entire project is huge. And when we modified the ISA, it means we need to modified all the code we have finished before. To solve these problems, in our design of Swift Assembler, we use GADL to describe our ISA to separate the flow of mapping from the code. The design flow is as Fig. 4.

4.1 The Description of ISA with GADL

In the design, the description of ISA contains the instruction description, type description, register description and many other descriptions.

Before all the descriptions defined, we firstly defined enumeration description to contains all the keyword which may be used in the next work.

- (enum
- (GR_E

- 'GR0' 'GR1' … 'GR31'
-)
- (VR_E
- ('VR0' 32) 'VR1' … 'VR15'
-)
- (BFEXT_E
- 'bfext' 'bfextu'
-)
- (…)
-)

After completing the enumeration type definition, we can go on our definitions of other descriptions.

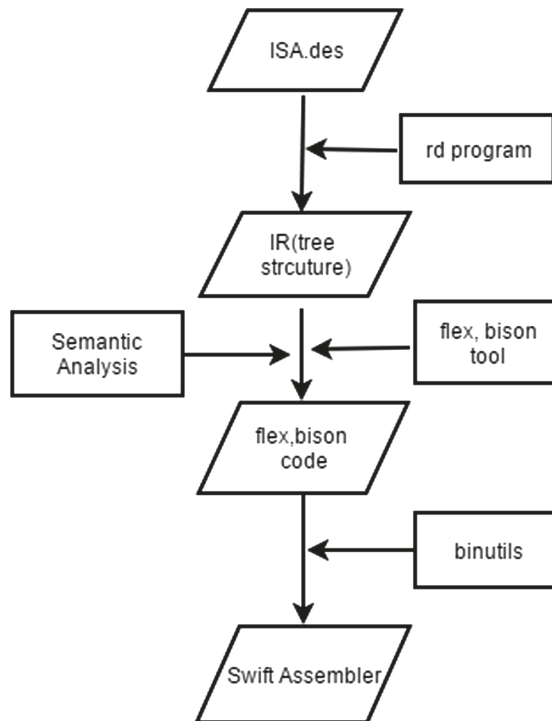

Fig. 4. The design flow of Swift Assembler

As mentioned above, the new DSP issues 8 instruction in parallel. So, we can define our instruction set with GADL like this.

- (instruction
- (top instrs
- (instrs

- (= i slots_8)
-)
-)
- (code i)
- (binary i)
- (vliw 1 0)
-)

In the description above, ‘code’ indicates the assembly parameter and the ‘binary’ is the machine code of the assembly parameter. The first number in the ‘vliw’ column is a flag to indicate whether VLIW mode is supported. And the second means whether this instruction is the last in the VLIW package. And then we can continue to define ‘slots_8’ as follows.

- (slots_8
- (= i slots0 empty)
- (= j slots1 empty)
- (…)
- (= y slots7 empty)
- (pack i j … y)
-)
- (slot0
- (= i NOP RET RTT JC JNC …)
- (code i)
- (binary ‘000’ i)
-)
- (…)
- (slot7
- (…)
- (code i)
- (binary ‘111’ i)
-)

As described above, we define 8 slots and specify the instruction that each slot can hold based on the hardware design. And next, we defined the assembly format for each instruction. To understand it easily, an example is provided.

- (BFEXT
- (= i BFEXT_E)
- (= rd GR_E)
- (= rs GR_E)
- (= bf0 imm5)
- (= bf1 bf5)
- (code i ‘ ‘ rd ‘ ‘ rs ‘ ‘ bf0 ‘ ‘ bf1)
- (binary ‘110’ ‘0’ rd ‘0’ rs bf0 bf1 ‘0’ i ‘01’)
-)

In this rule, we defined an assembly code "bfext rd rs bf0 bf1". This instruction is to achieve bit extraction. Similar to the examples given above, we can give many other definitions. All of them together describe the ISA with GADL.

4.2 The Semantic Analysis of GADL

Since GADL is a prefix expression, it is easy to create a LL1 read program rd. With the help of rd, we can convert from the text description to a tree structure.

- Class node {
- int type;
- Vector<node *> nodelist;
- String str;
- }

When the value in the node is a string type with quotes or a variable, the parameter nodelist is invalid. Besides, when the value in the node is a vector, the parameter str is invalid. Then we need to perform semantic analysis on the resulting tree structure. The most important step in the semantic analysis process is the rule expansion.

The rules of GADL is similar to the rules of nML [9, 10], and we also provide sub-rules to avoid rewriting the same rules. So, in the rule expansion, we need to fill each sub-rule into the parent rule that reference it. In this process, we firstly build a hash table (VarTable) to save reserved word. The key of each item in the table is a string, and the value is its corresponding attribute which is saved by UnfoldedList. This UnfoldedList is the rule we may need to expand. Figure 5 shows the rule expansion

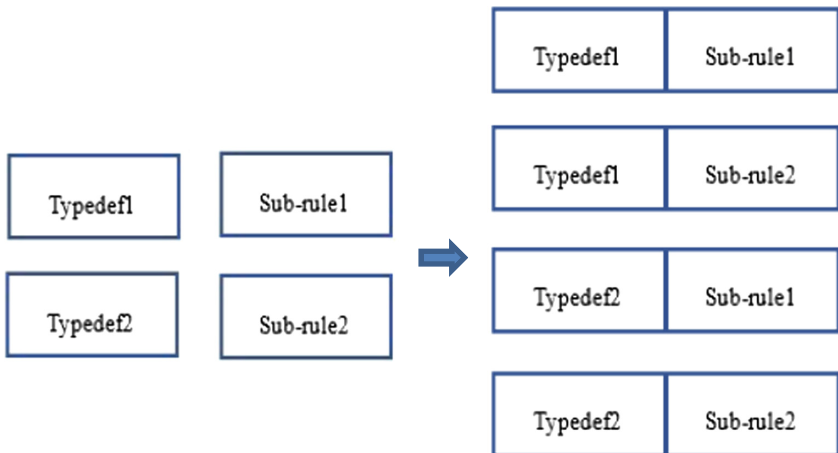

Fig. 5. Rule expansion

Since a rule may be expanded to contain a set of rules, we need to save it with a vector. Each rule is described by 7 sets of data, which are the expanded assembly description(code), binary(binary), variable(varName), the offset of the variable

in the assembly description(offsetInBinary), the width of the variable after expansion(varLen), the declared name of enumerated type(enumName) and the information of relocation(relocInfo). The structure is as follows.

- class UnfoldedList {
- vector <unfolded>lst;
- }
- class Unfolded {
- string rulename;
- vector<string>code;
- vector<string>binary;
- vector<string>offsetIncode;
- vector<string>offsetInbinary;
- vector<string>varName;
- vector<string>varLen;
- string enumName;
- vector<int>relocInfo;
- }

The pseudo code of the rule expansion is given below.

- Unfold(nodePtr)
- nodename = node->str
- if nodeName in VarTable
- do return VarTable[nodename]
- switch type of (nodePtr)
- case instruction
- ret = unfoldinstr(nodePtr)
- case enum
- ret = unfoldenum(nodePtr)
- case int_type
- ret = unfoldint_type(nodePtr)
- case address
- ret = unfoldaddress(nodePtr)
- VarTable[nodeName] = ret

The pseudo code above will continue to call the relevant instruction expansion algorithm, enumerate the expansion algorithm and other algorithms to complete the development of the GADL description language. Of course, we also need to do some work required by the hardware. For example, in our design of the new DSP, the VLIW supports 8 slots. Apart from the VLIW end flag, we only have two bits to correspond to 8 slots. To solve this problem, in the rule expansion we need to declare that both slot0 and slot1 correspond to '00'. Similarly, slot2 and slot3 correspond to '01' and so on. Because the instructions are in order, we can tell which slot of the two adjacent slots the instruction belongs to by the order.

4.3 Porting of the Underlying Toolset of Binutils

In this step, we deeply analyzed the workflow of the mips assembler and referred to the implementation code of mips assembler code. By using Flex [11] and Bison [12] tools, the code needed is generated easily. All the descriptions completed with GADL above are transferred to the flex and bison code. Also, the BFD library code of Binutils according to the address type used by the instruction set is generated at the same time. The example is as follows.

GADL:

- enum(
- J_E(
- 'jmp' 'jc' 'jnc' 'call'
-)
-)

Flex:

- "jmp" return TOK_62
- "jc" return TOK_63
- "jnc" return TOK_64
- "call" return TOK_65

Bison:

- J_E: TOK_62 {$$=(char*) "00"
- TOK_63 {$$=(char*) "01"
- TOK_64 {$$=(char*) "10"
- TOK_65 {$$=(char*) "11"

At last, the flex and bison code is transferred to the cpp file which can be identified by the Binutils tool set. With the help of the Binutils tool set, the Swift Assembler is compiled and generated.

5 Results and Verification

In the current design, there are 186 basic assembly instructions for the new DSP. In addition, the processor supports 1 to 8 variable-length VLIW instructions. The description file of instruction set has 860 lines. All remaining files are less than 6000 lines containing much code generated by the flex and bison tools. Compared with other assemblers, the work on programming is easily.

To verify the correctness of the Swift Assembler, assembly code containing all instructions is tested and passed. The machine codes generated run normally on the new DSP. Figure 6 shows the working process of SA.

```
!nop|nop||||||!
00000000  000000a0
!addi GR2 GR2 0xfc84|||||||!
20a42098
!|addic GR5 GR6 0xfc84||||||!
  24a451b8
!|||||||store8 GR7 GR8 0xd5!
              a01a72ec
```

Fig. 6. Assembly process of SA

6 Conclusion and Future Work

This article presents the design and implementation of Swift Assembler of a new DSP with high performance. By using the GADL to describe the ISA of the new DSP, we present the semantic analysis of GADL and introduce the tools of flex, bison and Binutils. At last, we illustrate the actual working process of SA. Compared with traditional assemblers, SA has less coding and more flexible architecture. This bring great convenience to other colleagues in our team.

In the future, we will continue to optimize the design of assembler. And based on the GADL, we will go on our research for disassembler, simulator and debugger.

Acknowledgment. The authors thank the editors and the anonymous reviewers for their invaluable comments to help to improve the quality of this paper. This work was supported in part by the National Natural Science Foundation of China under Grant Nos. 61831018, 61571329 and 61631017, and Guangdong Province Key Research and Development Program Major Science and Technology Projects under Grant 2018B010115002.

References

1. Ji, X., Huang, K., Jin, L., Tang, H., Liu, C., et al.: Overview of 5G security technology. Sci. China (Inf. Sci.) **61**(08), 107–131 (2018)
2. GNU Binary Utilities. http://www.sourceware.org/binutils/
3. Clements, P.C.: A survey of architecture description languages. In: Proceedings of the Software Specification and Design. IEEE Computer Society, April 1996. https://doi.org/10.1109/iwssd.1996.501143
4. Shen, J.: A research on processor architecture description languages and implementation of tool-chain generation (unpublished)
5. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA
6. Lin, T.J., Chen, S.K., Kuo, Y.T., et al.: Design and implementation of a high-performance and complexity-effective VLIW DSP for multimedia applications. J. Sig. Process. Syst. **51**(3), 209–223 (2008)
7. Hu, Y., Chen, S.: Preprocessing scheme of intelligent assembly for a high performance VLIW DSP. In: Second International Conference on Cloud & Green Computing (2013)

8. Hadjiyiannis, G., Hanono, S., Decadas, S.: ISDL: an instruction set description set processors for retargetability. In: Proceedings of the 34th Annual Design Automation Conference, pp. 299–302. ACM (1997)
9. Freericks, M.: The nML machine description formalism
10. Fau, A., Van Praet, J., Freericks, M.: Description instruction set processors using nML. In: Proceedings of the European Design and Test Conference, ED&TC 1995, pp. 503–507. IEEE (1995)
11. The Fast Lexical Analyzer. http://flex.sourceforge.net/
12. GNU Bison. http://www.gnu.org/software/bison/