Porting LLVM Compiler to a Custom Processor Architecture Using Synopsys Processor Designer

Hyungyun Jung, Jangseop Shin, Ingoo Heo, Yunheung Paek
Dept. of Electrical and Computer Engineering, Seoul National University
e-mail: hgjung, jsshin, igheo@sor.snu.ac.kr, ypaek@snu.ac.kr

Abstract

Application specific instruction-set processor (ASIP) is a suitable design choice for system designers who seek both flexibility to handle various applications in the domain together with the performance. Successful development of an ASIP, however, requires a software development kit (SDK) to be provided along with the processor. Synopsys Processor Designer is an ASIP development tool, which takes as input a set of files written in a high-level architecture description language called LISA (Language for Instruction Set Architecture), and generates SDK as well as RTL. Recently, they have added support for the generation of LLVM compiler backend, though some manual work is required. In this paper, we introduce some details in porting LLVM compiler to a custom processor architecture in Synopsys Processor Designer.

1. Introduction

Nowadays, with the development of integrated devices, designers pursue not only higher performance and lower power consumption, but also the flexibility of design. In this case, traditional fixed hardware blocks, which cannot fulfill the latter desire, are inadequate. Application specific instruction-set processor (ASIP), a kind of processor used in system-on-a-chip design, can be a good alternative. The ability to offer flexibility through software reprogrammability while limiting overhead makes ASIP the ultimate trade-off between performance, power consumption and flexibility.

Successful development of an ASIP, however, requires a software development kit (SDK) to be provided along with the processor. Synopsys Processor Designer (SPD)[1] is an ASIP development tool, which takes as input a set of files written in a high-level architecture description language called LISA (Language for Instruction Set Architecture), and generates SDK as well as RTL. Recently, they have added support for the generation of LLVM compiler backend which will generate base files for porting LLVM backend to the custom processor. But meanwhile, there still remains a lot of details to be provided by manual work.

In this paper, we introduce some details about porting LLVM compiler backend to the template VLIW processor model provided by SPD as an example. The remaining sections of this paper is organized as follows: Section 2 briefly introduces the background on LLVM compiler and its major compilation stages, Section 3 mainly talks about the porting details, Section 4 will show the result, and Section 5 will finally conclude this paper.

2. Background

LLVM compiler infrastructure[2] is an open source compiler platform designed as a set of reusable libraries with well-defined interfaces, which enable various subprojects to be easily built on and architecture backend to be added without much effort. Together with its BSD-style license, which keeps its users from the responsibility of disclosing their work, it has become one of the popular open source compiler framework among industrial as well as academic communities.

LLVM backend compilation flow consists of numerous stages, but it can be summarized into the following three important phases: Instruction selection, Register allocation, and Instruction scheduling.

Instruction selection is a phase that translates the machine independent intermediate representation (IR) into initial target specific code. Register allocation is the process of allocating real register numbers to virtual registers, adding spill code if necessary.

In instruction scheduling phase, instructions are reordered to reduce execution time, or in the case of statically scheduled architectures like VLIW, instructions are scheduled at appropriate cycle and slot for fast and correct execution of the program.

For the purpose of getting the compiler to generate “working” code, not “optimized” code, the instruction

selection phase is the most important, since corresponding machine code should be specified for each IR pattern. Also, for statically scheduled processors, instruction scheduling part needs some manual work, since instruction dependence relation must be specified.

3. Porting LLVM compiler to template VLIW architecture

When we generate an LLVM backend, SPD will generate backend codes for an example RISC architecture. The workload of generating codes by manual work depends on how much the difference in our target architecture and the example RISC architecture. In this part we will mainly talk about the manual work done in instruction selection phase and instruction scheduling section phase.

3.1. Instruction selection

LLVM employs a SelectionDAG instruction selector (generated from the target description '*.td' files) in this phase and after this phase the SelectionDAG will be destroyed. The SelectionDAG is a directed-acyclic-graph with its nodes are SDNode class instances. The operation code of SDNode indicates what operation the node represents and the operands to the operation.

Instruction selection phase in LLVM compiler is mainly constructed by three steps: Type legalization, Operation legalization, and Instruction selection.

3.1.1. Type legalization

By operating on the SelectionDAG nodes, type legalization removes the types which target machine does not support. Basically there are two ways of the type legalization: promoting (convert small type into large one) and expanding (convert large type into small one).

Since our target VLIW template matched the example RISC implementation very well, there was no need to make modifications in this step.

3.1.2. Operation legalization

Similar to the type legalization, by operating on the SelectionDAG nodes, operation legalization removes the operations which target machine does not support. To distinguish if the operation is supported by the target machine, in the DAG->Legalize () function, we set one of the following three flags to each circumstance: Legal, Expand, and Custom.

There is a list of operation node definitions provided (figure 1), and what we need to do in this step is to find out proper pattern for target architecture. If we can find a matching pattern of operations for target architecture, we simply use these operations and set the flags of them to Legal, otherwise we need to set flags to Expand or Custom.

For example, conditional branch selection operation in the example RISC architecture, was implemented by node CMPCC. The pattern of node_cmpcc (figure 1, line 1), which uses a temporary register CC to store the conditional flag true or false, does not match our target VLIW architecture which does not contain a CC register. Then we found out that another node SELECT perfectly matches the pattern of our target architecture. So we comment the node CMPCC and change the flag of ISD::SELECT to Legal (figure 2, the flag was Expand in example RISC architecture). In the same way we take usage of node SETCC and also set its flag to Legal.

```
//def SDT@NS@_CMPCC  : SDTypeProfile<1,3, [SDTCisInt<0>, SDTCisSameAs<1,2>, SDTCisVT<3, OtherVT>]>;
def SDT@NS@_BEQ    : SDTypeProfile<0,1, [SDTCisVT<0, OtherVT>]>;
def SDT@NS@_SELECT  : SDTypeProfile<1,2, [SDTCisSameAs<0, 1>, SDTCisSameAs<1, 2>]>;
```

(Figure 1) Provided Operation Nodes.

```
setOperationAction(ISD::SELECT_CC      , MVT::Other, Expand);
setOperationAction(ISD::SELECT_CC      , PD_MASTER_MVT,  Custom);
setOperationAction(ISD::SELECT         , PD_MASTER_MVT,  Legal);
setOperationAction(ISD::SETCC          , PD_MASTER_MVT,  Legal);
```

(Figure 2) setOperationAction with three kinds of flags.

3.1.3. Instruction selection

This step will map the SelectionDAG operations to the target instructions by doing the pattern matching, and translate machine independent SelectionDAG into target specific instruction DAG.

As the example shown in figure 3, in template VLIW architecture, this step will do the pattern matching which maps the SelectionDAG node "add" operation (machine independent) into the "ADD" instruction (target specific) which was generated manually.

```
ISEL: Starting pattern match on root node: 0xc7bd4a4: i32 = add 0xc7bd1e8, 0xc7bd530 [ORD=6] [ID=16]

  Initial Opcode index to 226
  Match failed at index 232
  Continuing at 267
  Morphed node: 0xc7bd4a4: i32 = ADD 0xc7bd1e8, 0xc7bd530 [ORD=6]

ISEL: Match complete!
```

(Figure 3) Instruction Selection.

3.2. Instruction scheduling

For static scheduling architecture like VLIW, where complex hazard detection hardware logic is absent, compiler is responsible for positioning instructions in order to prevent resource conflicts and data hazards. Thus, low level architecture information like, in which VLIW slot can execute which instructions, in which

pipeline stage the source operands are read and the destination operands are written for each instruction, should be provided to the compiler. By using these information, compiler can implement the schedule correctly. For this purpose, SPD provides scheduler template at the very last stage of the compilation flow. The scheduler template uses three tables, each corresponding to RAW, WAR, WAW hazard, to calculate the latency between any two instructions.

Assume that all the instructions read their register operands in the second stage of the pipeline, and the result of ADD instruction is available in the next cycle, except the result of MUL instruction is available after two cycles. Also assume that the hardware includes full bypassing logic, then the instruction using the result of ADD can be scheduled at the next cycle, but the instruction using the result of MUL should be scheduled after two cycles.

Let's say the defined operands in ADD instruction are of class 0 and those for MUL instruction are of class 1. To find out how many cycles the instruction should be apart from defining instruction, the scheduler consults RAW latency table, using the class number as the index. The table is actually two dimensional, covering the case where some instruction reads operands at different pipeline stages from others. For RAW hazards, defined operands are called 'source' and the operands using the value are called 'sink'. For WAR hazards, read operands are called 'source' and def operands are called 'sink'.

Source / Sink	Class 0	Class 1	Class 2
⋮	⋮	⋮	⋮
Class n	1	2	3
⋮	⋮	⋮	⋮

(Figure 4) Latency Table.

Now, for each instruction, we only need to provide information about which operands belong to which class, when read operands are used as source/sink and when def operands are used as source/sink. Then the scheduler can compute the latency between any two instructions, and also compute the correct schedule for instructions.

Usually available functions for different VLIW slots are not the same. For example, operations that change program flow are only permitted in one of the slots, since it doesn't make sense to have multiple branching operations in one VLIW word. To take this into account, we added a getValidSlotMask () function to tell the scheduler in which slots each instruction can be scheduled.

3.3. Others

To complete LLVM backend, other minor code pieces should be implemented, including calling convention, frame lowering, expanding pseudo instructions, and assembly printing.

Calling convention for the architecture should be described in XXCallingConv.td file, which provides information about how function arguments are passed to the callee, and how the callee returns value to the caller.

Frame lowering provides information about which instructions should be added at the start/end of the function to set up/destroy stack frame for the function.

Expanding pseudo instruction transforms temporary instructions into real machine instructions. Instead of mapping all IR code directly into real machine code in instruction selection stage, some instructions are mapped to pseudo code and later expanded into real code. One example is loading large immediate into a register, which cannot be done with one instruction due to the restricted immediate field in instruction word. It is mapped to the pseudo instruction at first and later will be replaced by correct sequence of real instructions which usually consist of loading upper and lower part of the register.

Finally, assembly printing part should be fixed so that VLIW words are printed according to the syntax pattern expected by the assembler.

4. Result

The test case used in porting LLVM compiler to template VLIW architecture is called Annotate-From-Scratch (AFS) build test case, a list of C test codes to check if the ported LLVM compiler is working correctly. The AFS test case will check the correctness of totally 50 C test codes including ALU operations, load/store operations, branch operations and so on.

After the manual work, we generated all the target specific machine instructions for corresponding machine independent operations, modified the target description XX.td files and the scheduler files to match the target VLIW architecture. As a result, after manual generations and modifications, all the 50 AFS test codes have been passed through successfully.

5. Conclusion & Future Works

In this paper, we first made an introduction to the ASIP and SPD, then briefly mentioned the background

of LLVM and LLVM compilation flow, then mainly talked about the details of porting LLVM compiler to template VLIW architecture, and finally stated the result. By generating the target specific machine instructions and modifying the target description XX.td files and the scheduler files, we made the AFS test case of LLVM compiler to template VLIW architecture built successfully.

With the experience of this porting LLVM compiler to a custom processor architecture project, we are going on with developing an ASIP for a hand gesture recognition system, and porting a corresponding LLVM compiler backend for it.

6. Acknowledgement

This research was partly supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning (MSIP) / National Research Foundation of Korea (NRF) (Grant NRF-2008-0062609), the IT R&D program of MSIP/KEIT [K10047212, Development of homomorphic encryption supporting arithmetics on ciphertexts of size less than 1kB and its applications], Business for Cooperative R&D between Industry, Academy, and Research Institute funded by Korea Small and Medium Business Administration in 2014, the Brain Korea 21 Plus Project in 2014 and IDEC.

7. Reference

[1] Synopsys Inc., Synopsys Processor Designer, http://www.synopsys.com/Systems/BlockDesign/ProcessorDev/Pages/default.aspx

[2] C. Lattner, V. Adve, “LLVM: a compilation framework for lifelong program analysis & transformation,” *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, pp. 75-86, March 2004.