GCC based Compiler Construction for Compact DSP32[1)]

Myeongjin Cho[1], Hokyoon Lee[1], Giang Nguyen Thi Huong[1], Seon Wook Kim[1],
Youngsun Han[2], Jungyoung Um[3]
[1]School of Electrical Engineering, Korea University
[2]School of Electrical Engineering, Kyungil University
[3]Zaram Technology
e-mail: linux@korea.ac.kr

Abstract

Very Long Instruction Word (VLIW) executes multiple instructions in parallel. In order to exploit higher performance, i.e., higher parallelism, VLIW compiler groups as many instructions into one word as possible. In this paper, we show how to construct a VLIW C compiler based on GCC for CDSP32 (Compact Digital Signal Processor 32-bit) which is an embedded DSP processor to issue two instructions in one VLIW. Also, we evaluated the compiler on EEMBC benchmark; the experiment result showed that the total number of dynamic instructions of the VLIW compiler was reduced by 18% on average over without VLIW instruction scheduling.

1. Introduction

A hardware performance in modern embedded processor design is one of the most important issues because signal processing power becomes critical concern in multimedia devices. To achieve higher performance, many chip venders have adopted hardware techniques to their own processors. Especially, the VLIW architecture [1] is an attractive solution in embedded processor design, because the VLIW architecture takes advantage of instruction level parallelism (ILP) in applications to execute instructions in parallel. Nevertheless, the VLIW architecture does not need complex hardware to schedule instructions, since the compiler provides VLIW instructions that is already scheduled considering ILP.

ZARAM [2] recently developed a 32-bit Compact DSP processor that supports the VLIW execution. The CDSP32 processor can execute two independent instructions at the same time. In this paper, we port GNU C Compiler [3] to the CDSP32 processor and apply an algorithm of VLIW instruction constructing. We also evaluate our work with EEMBC benchmark on the CDSP32 simulator and measure the total number of dynamic instructions. As a result, we could achieve 18% of performance enhancement in terms of the total number of dynamic instructions over without VLIW approach.

The rest of this paper is organized as follows. Section 2 explains the compiler construction for CDSP32. Section 3 discusses the VLIW constructing algorithm in detail. We show performance results in Section 4 and conclude this paper in Section 5.

2. Compiler Construction for CDSP32

The CDSP32 processor is a 32-bit DSP core capable of operating at 250MHz speed. It uses 16kB 4-way set-associative data/instruction caches and supports 8, 16, 32, 64-bit data size. The CDSP32 processor has 14 general-purpose registers, each of 32 bit width and two instruction issue logics for VLIW.

The CDSP32 processor supports arithmetic, logical, memory load/store, branch, and move operations. For GCC-based compiler construction, all the possible operations should be described as GCC IR (Intermediate Representation) in the compiler's MD (Machine Description) file. In addition, all of logical registers should be defined as register classes according to their usages. The register classes are used as a constraint in the patterns.

Figure 1 illustrates an example of a *single integer move* pattern in the MD file. The GCC emits assembler mnemonics according to the constraint of operands. In Figure 1, we assigned *mov* op-code to register-to-register copy, and *ld32* op-code to the case of memory-to-register load. Also, we added two attributes to each instruction; pattern type and VLIW information. The pattern type describes the kind of corresponding instruction, and the VLIW information explains which slot the instruction can go; slot0 or slot1. The VLIW constructing algorithm will be explained in the next section.

```
// Single Integer MOVE Pattern
(define_insn "*movsi"
 [(set
 (match_operand:SI 0 // Destination operand
       "nonimmediate_operand"  "=r,r,r,a,m")
 (match_opernad:SI 1 // Source operand
       "general_operand"       " r,m,i,r,r"))]
 ""
 "@
    mov     %r0, %r1    // Register to Register copy
    ld32    %0,  %m1    // Load from memory
    ld      %0,  %1     // Assign immdiate const to register
    wrr     %0,  %r1    // Register to SP register copy
    st32    %m0, %r1    // Store to memory
 "
 [(set_attr "type"      "move, load, load, move,store")
  (set_attr "insn_type" "vliw1,vliw0,vliw1,none,vliw0")])
```

Figure 1. An example of a single integer move pattern in the MD file

3. Constructing VLIW Instruction

The CDSP32 processor can execute at most two instructions at a time by grouping them into a VLIW

[1)] This paper has been supported by ITEP System IC2010(10030565-2010-04).

instruction. However, several restrictions should be carefully handled to form the VLIW instruction: 1) one memory port is supported, two memory instructions cannot be grouped; 2) branch instructions such as jump and function call should be excluded; 3) instructions such as *call*, *pop*, and *push* should be put at the first slot of VLIW.

Before grouping instructions into a VLIW instruction, the compiler should check whether instructions can be executed in parallel. The GCC's instruction scheduler checks the dependency. At each cycle, the scheduler provides a list of parallel instructions that can be executed together in one cycle. From the given list, the VLIW constructor chooses one or two instructions.

The detail of VLIW grouping can be explained as follow: 1) The CDSP32 compiler searches and identifies candidate instructions from the independent list to be put into slot0 and/or slot1 of VLIW. 2) The compiler tries to fill VLIW slots with the candidate instructions while considering the restriction rules of the CDSP32 processor; a pair of slot0 and slot1 instructions constructs one VLIW instruction. 3) The compiler marks the matching instructions into a linked list and reorders the instruction sequence if necessary: the instruction at slot0 must be put right before the instruction at slot1 in the independent list. At the end of a basic block scheduling, the CDSP32 compiler inserts a special instruction to mark the starting point of a VLIW in the RTL instruction sequence so that the CDSP32 processor can recognize them at run time. The VLIW constructing algorithm is as shown in Figure 2.

```
/* list provided by GCC scheduler to identify instruction
   to execute within one cycle. */
for_each independent_insn_list {

  for_each insn in independent_insn_list{
  // slot0, slot1 or none (cannot used in VLIW)
    type = recognize_VLIW_type_of_insn(insn);
    new_vliw = find_matching_VLIW_slot(type,
               independent_insn_list, insn);
    if(new_vliw) {
      vliw_link_list += new_vliw;
      reorder_DSP_instructions(new_vliw, independent_insn_list);
      break;
    }
  }
}

for_each link in vliw_link_list {
  // insert a vliw marker before slot0 DSP
  insert_vliw_marker(link);
}
```

Figure 2. Pseudo code of VLIW constructing algorithm.

Since the CDSP32 processor can execute only one VLIW instruction at a time, the algorithm exits the innermost loop after finding a VLIW instruction. Table 1 shows target HOOKs to enable the VLIW construction.

Table 1. Target HOOKs

TARGET_SCHED _REORDER	Main VLIW instruction constructing
TARGET_SCHED _REORDER2	The order of instruction inside the list can be changed after GCC scheduler process, reorder the VLIW slots if necessary
TARGET_SCHED _FINISH	Insert marker instruction to mark VLIW groups
TARGET_SCHED _VARIABLE_ISSUE	Helper function

4. Performance Evaluation

Figure 3 shows an example of C and assembly codes when the VLIW constructing algorithm is applied. We can see two *vliw* instructions each of which are followed by two independent instructions. When the CDSP32 processor fetches a *vliw* instruction, the subsequent two instructions (i.e. *rdr*, *ld*) are grouped into a VLIW instruction and executed in parallel.

For our performance evaluation (in terms of the number of dynamic instructions), we used EEMBC benchmarks [4] on the CDSP32 simulator. Figure 4 shows the number of dynamic instructions on *non-vliw* and *vliw* compiler. The *non-vliw* and *vliw* indicate versions before and after applying the VLIW constructing algorithm, respectively. The result was normalized to the number of dynamic instructions of the *non-vliw*. We could reduce the number of dynamic instructions by 18% on average in the *vliw*.

Figure 5 shows the ratio of instructions combined into VLIW instructions among total instructions. We could match the VLIW instructions about 36% on average. This result represents that our VLIW constructing algorithm is working well despite the processor's hardware restrictions.

```
int sum=0;
int main(void) {
  int i;
  for (i=1;i <= 10; i++)
  {
    sum += i;
  }
  return sum;
}
```

```
_main:
    #r1 r7
    pushx   0x82
    vliw
    rdr     r7, sp
    ld      r1, _sum
    ld32    r1, *r1
    add     r8, r1, 55
    vliw
    ld      r1, _sum
    mov     r0, r8
    st32    *r1, r8
    #r7 r1
    popx    0x82
    ret     UNC
```

Figure 3. Example of C and assembly codes.

5. Conclusion

The VLIW architecture is a popular solution for high performance DSP processor, because it can execute multiple instructions in parallel without complex hardware. However, the compiler has to determine the VLIW instructions that can be run in parallel. In order to achieve good performance, the compiler has to schedule instructions well to take advantage of as much ILP as possible.

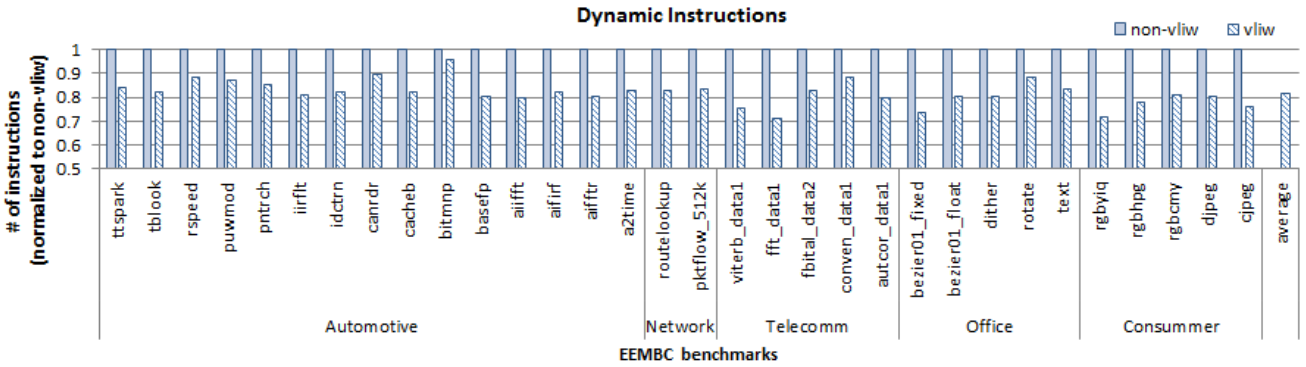

Figure 4. The number of dynamic instructions on EEMBC benchmarks.

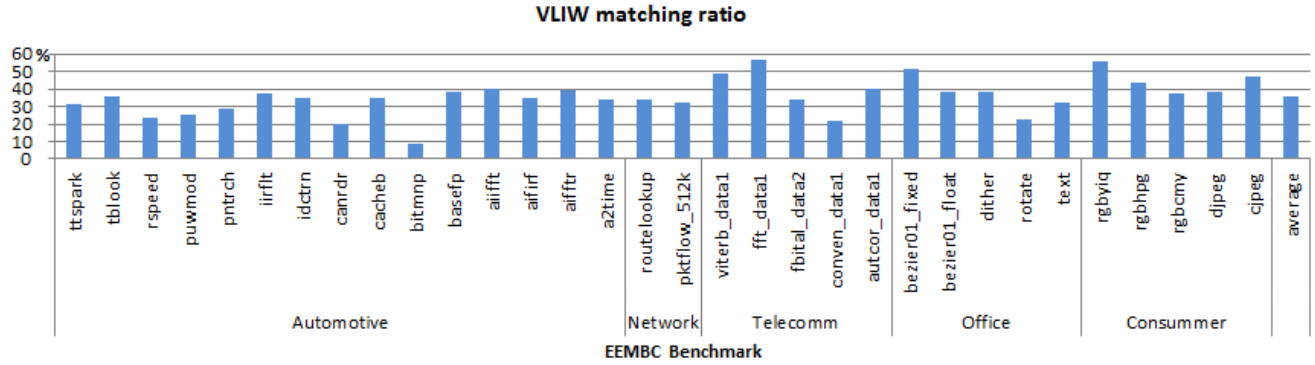

Figure 5. The matching ratio of VLIW patterns on EEMBC benchmarks.

In this paper, we built the Compact DSP32 compiler based on GCC, and applied VLIW constructing algorithm for supporting VLIW instructions. Compared to the *non-vliw*, the *vliw* reduced the number of dynamic instructions by about 18% on average for EEMBC benchmarks. In addition, we could extract the VLIW instructions about 36% on average.

References

[1] Joseph A. Fisher. 1983. Very Long Instruction Word architectures and the ELI-512. In *Proceedings of the 10th annual International Symposium on Computer Architecture* (ISCA '83). ACM, New York, NY, USA

[2] Zaram Technology, http://www.zaram.com

[3] GNU Compiler Collection, http://gcc.gnu.org

[4] EEMBC benchmark suites, http://www.eembc.org