Current Science
CS 6 (1), 5724-5741 (2026)
Received 10 November 2025 | **Accepted 25 November** 2025 | **Online Available 12 January** 2026
https://doi.org/10.52845/CS/2026-6-1-7

Current Science

Current Science

OPEN ACCESS

ISSN (O) 2795-8639

Original Article

A Decoding Algorithm and its Optimization for Mixed-Length Instruction Sets with Non-Uniform Encoding

Ye Tian[1], Shaofeng Wu[1], Longzhen Chen[1], Mao Zhang[1]

[1]DSP Research Office, China Electronics Technology Group Corporation 58 Research Institute, Wuxi, 214000, China

***Corresponding Author: Ye Tian**

Abstract:
The software decoding algorithm of instruction sets is widely applied in simulators, debuggers and reverse engineering. Current research predominantly focuses on fixed-length instruction sets and variable-length instruction sets, with limited studies addressing non-uniformly encoded mixed-length instruction sets. This paper introduces a two-level reading strategy for machine instructions. With the help of a decoding framework, the instruction boundaries of mixed-length instruction sets are identified and machine instructions in executable files are parsed. Following this, a three-stage decoding algorithm is proposed. In the first stage, preliminary decoding identifies all matching pattern sets of instructions. The second stage determines the optimal matching pattern and corresponding instruction length. The third stage resolves encoding conflicts specific to 32-bit instructions. To further enhance the algorithm, a priority-based optimization method is designed and applied for the above algorithm. The proposed approach is tested on common algorithms and standard library functions of DSP. The experimental results demonstrate that the optimization method effectively improves efficiency.

Keywords: Non-uniformly encode, Mixed-length instruction, Instruction decode, Optimization

1. Introduction

Instruction set decoding refers to identifying byte streams, mapping legal instructions to specific instructions, translating to obtain information such as operands [1], and mapping illegal byte streams to invalid instructions. It is mainly used in CPU software simulators, compilers, debuggers, security analysis, binary rewriting, and reverse engineering [2]. Instruction decoding is a necessary key step in the tool chain. For compiler development of embedded chips, it is cumbersome and time-consuming to burn executable files to the chip each time, and batch testing is often done with the help of software simulators. For debuggers, the stack frame and PC are maintained locally, and the length of each instruction needs to be known to facilitate the next step of updating the PC, which is essentially a kind of decoding. In addition, instruction set simulators simulate the target machine program by simulating the execution effect of each instruction on the target processor, and play an important role in the design, verification, and application of microprocessors. The accuracy of instruction set decoding is the basic premise of subsequent work [3][4]. The instruction set was originally designed as a fixed-length instruction set, and its instruction decoding is simple. Later, variable-length instruction sets represented by CISC instruction sets were widely used in the desktop field, and the decoding research on variable-length instructions was also quite sufficient. However, mixed-length instruction sets

are not widely used, primarily in the DSP field, and the studies on decoding this type of instruction set are relatively scarce.

The contributions of this paper can be summariezed in the following three aspects:

1. A general framework for parsing machine instruction of mixed-length instruction sets is introduced, providing a flexible foundation for handling such architectures.
2. A novel software decoding algorithm is proposed to support implement of mixed-length instruction sets;
3. The proposed software decoding algorithm is further optimized to improve efficiency;

Instruction decoding has been developed for decades and is widely regarded as a mature technology. In its early stages, instruction sets were typically fixed-length，making decoding relatively straightforward to implement. Tor E. Jeremiassen proposed a method for generating an instruction-level simulator by defining an instruction description file. The decoder relied heavily on a large number of conditional if judgments [5]. However, this approach had notable limitations, including reduced readability and maintainability due to the extensive use of conditional logic.

Other software decoding algorithms include table lookup, mask matching, and decision tree algorithms [6]. The table lookup method involves storing all instructions in the instruction set into a hash table, enabling O(1) complexity for search operations. In this method, the opcode serves as the key, while instruction-specific processing function pointers are stored as the corresponding values. Upon receiving an instruction, it is matched against entries in the hash table sequentially. This algorithm offers advantages such as simplicity in implementation and high execution efficiency. However, it is primarily suitable for relatively simple instruction sets and poses challenges in accommodating instructions with complex operand structures.

Another commonly used implementation approach is the mask matching method. This method relies on the mask and matching code for each instruction. The mask extracts specific bits from the original instruction, typically the opcode encoding bits, and compares them with the matching code. If a match is not found, the process iterates to the next pair of mask and matching code [7][8][9].The mask matching method offers advantages such as straightforward implementation, high execution efficiency, and robust support for fixed-length instruction sets. However, it presents challenges for variable-length instruction sets, as it is difficult to determine whether the current instruction is complete, or whether it represents a 32-bit or 16-bit instruction. Additionally, in instruction sets with complex operand support, it is necessary to define individual masks and matching codes for each operand of every instruction, resulting in significant implementation effort.To address this, the regular decoding algorithm can simplify the process by extracting the operand decoding logic for a class of instructions, thereby enabling reuse. Furthermore, [10] proposed the ISA_ML language, which allows the visual description of microprocessor architectures and the generation of instruction decoders. However, its applicability is limited to 32-bit fixed-length instructions.

There is extensive research on decoder generators in the academic community, with most studies focusing on generating decision trees based on specific description languages. Theiling proposed a decision tree model for decoding, where the input data is checked for valid pattern bits at each step, then partitioned into subsets based on the valid bit data, with the process continuing recursively until a leaf node is reached [11]. For non-orthogonal instruction sets, Fournel et al. introduced two automatic decoder generation strategies based on Binary Decision Diagrams

(BDD) to represent instruction encodings, effectively addressing complex scenarios [6]. Tadros conducted a comparative study of various decision tree generation algorithms, considering factors such as tree size and depth, and developed a cost model to optimize decision tree construction [12].

Testing decoders and decoding algorithms is also a critical area of research. Although studies specifically targeting decoding tests for mixed-length instruction sets are limited, research on decoding tests for x86 instruction sets offers valuable insights. Reference [13] proposed a method to test decoders by identifying discrepancies between software and hardware decoders, uncovering numerous errors in widely used decoders without relying on hardware implementations. For testing disassemblers, reference [14] presented a feedback-based depth-first search (DFS) algorithm, which achieved high coverage and identified many defects in the Capstone disassembler.

1. Analysis of Instruction Sets and Encoding Characteristics

1.1. Executable File Analysis

After being translated by the compiler, high-level language code is converted into machine instructions and stored in a binary executable file. Common executable file formats include the Common Object File Format (COFF) and the Executable and Linkable Format (ELF). Among these, the elf file format is currently a more common format and has been more widely promoted.

An ELF file is composed of several key components, including the ELF Header, Program Header Table, Section Header Table, and Section Data [15]. The ELF Header contains fundamental file information, such as the Magic Number (used to identify the file as an ELF file), file type, target architecture, and endianness. The Program Header Table specifies the memory layout of the executable file, detailing the address and size of each segment to facilitate dynamic linking and program loading. The Section Header Table provides metadata for each section, such as its name and size. Finally, the Section Data contains the actual content of the program, with the .text section, for example, storing the executable code of the program. This paper focuses primarily on the content of the .text section, as it contains the machine instructions critical for the decoding process.

1.2. Instruction Set and Encoding Space

The instruction set serves as a critical interface layer between computer hardware and software, defining how the upper-level software interacts with and operates the hardware. To execute specific operations, developers must provide instructions in a format that the machine can interpret. As the sole programming language directly understood by the machine, the instruction set also acts as the foundational interface for tools like assembly languages and compilers. Figure 1 illustrates this relationship, highlighting the instruction set's role as the bridge between hardware and software development.

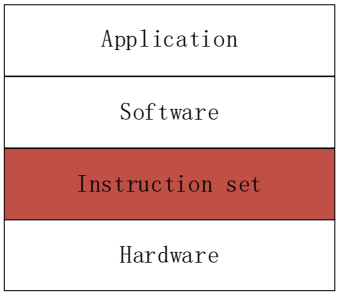


Fig.1 Instruction set and hardware and software hierarchy

There are several key concepts related to instruction sets that require clarification. The encoding space encompasses all possible binary encodings of instructions and determines the maximum number of instructions that can be defined within an architecture. For instance, a 16-bit instruction set has an encoding space size of 2^{16}, allowing up to 2^{16} distinct encodings. The instruction set space, on the other hand, refers specifically to the subset of encodings that the machine can recognize and execute. This space can be further divided into the public instruction set space and the private instruction set space. The public instruction set space includes instructions officially released by chip manufacturers for user applications, whereas the private instruction set space typically contains instructions reserved for internal chip testing or proprietary functions. Finally, the reserved space represents the portion of the encoding space not currently allocated to the instruction set space, often reserved for potential future instruction set extensions [16].

1.3. Classify Instruction Sets by Instruction Set Length

According to the instruction length in the instruction set, the instruction set can be roughly divided into fixed-length instruction set, variable-length instruction set and mixed-length instruction set.

Table 1 Comparison of different instruction sets.

Architecture	Instruction length/byte	type
x86	1-15	Variable length
RISC-V32	4	Fixed length
TIC28x	2 or 4	Hybrid length

A fixed-length instruction set refers to an instruction set in which all instructions have a uniform length. The primary advantage of this design is that instruction boundaries are explicitly defined, allowing hardware to efficiently decode single instructions and facilitating the simultaneous decoding of multiple instructions. Additionally, this predictability enables processors to allocate pipeline resources more effectively, thereby improving instruction-level parallelism (ILP). However, fixed-length instruction sets also have notable drawbacks. For simple instructions, unused bits are often filled with padding to maintain a consistent instruction length, leading to an increase in instruction size and inefficiency in encoding utilization. Furthermore, due to the constrained bit width, fixed-length instruction sets typically exhibit high encoding space utilization, leaving limited reserved space for future extensions and reducing scalability. Prominent examples of fixed-length instruction sets include RISC-V [17] and MIPS [18].

A variable-length instruction set refers to an architecture where instruction lengths vary within a defined range, determined by the functional requirements or operational complexity of each instruction. Simple instructions use shorter encodings, while more complex instructions require longer encodings, providing greater flexibility. This design eliminates the need for padding unused bits, resulting in higher instruction density. However, variable-length instruction sets present challenges: instruction boundaries are ambiguous, making decoding more complex, and the exact location of the next instruction is difficult to predict. A typical example of a variable-length instruction set is x86 [19].

In contrast, a mixed-length instruction set combines features of both fixed-length and

variable-length instruction sets. It typically consists of a limited number of fixed-length instruction types, such as 16-bit and 32-bit instructions coexisting. Compared to fixed-length instruction sets, mixed-length instruction sets avoid the inefficiency of padding unused bits, resulting in better encoding efficiency. However, determining instruction boundaries remains more complex than in fixed-length sets. Compared to variable-length instruction sets, mixed-length instruction sets provide clearer instruction boundaries but at the cost of slightly lower code density. Overall, mixed-length instruction sets strike a balance between code density and ease of boundary determination, positioning themselves between fixed-length and variable-length instruction architectures.

In high-performance or memory-constrained scenarios, variable-length and mixed-length instruction sets may arrange instructions of varying lengths consecutively at different addresses. This can lead to address misalignment, which complicates instruction fetching and decoding, ultimately resulting in a decrease in execution efficiency.

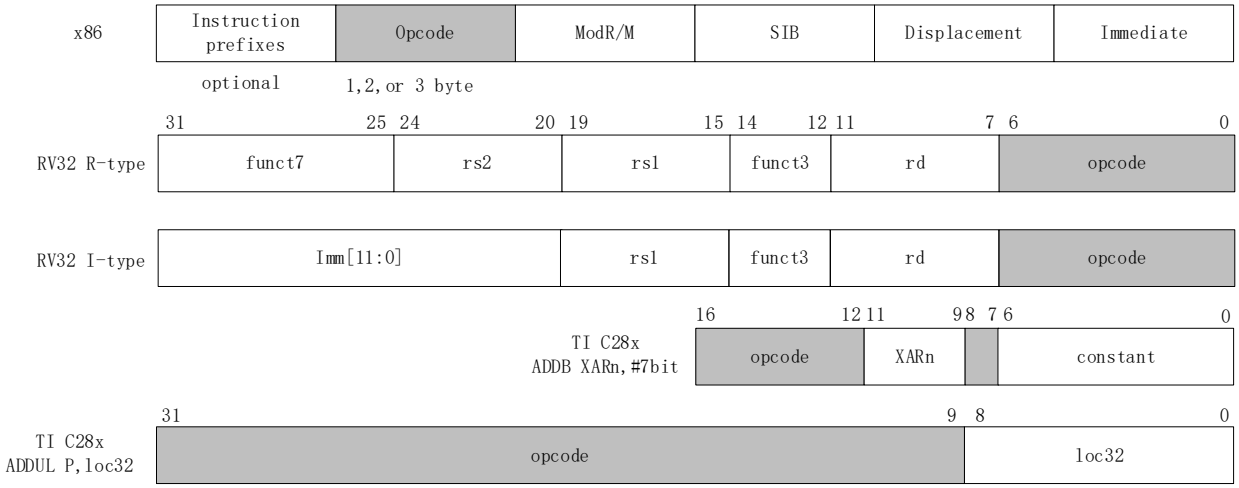


Fig.2 Length and encoding format of different instruction sets

1.4. Classify Instruction Sets by Encoding

Common unified encoding instruction sets include Aplpha, MIPS and RISC-V. The instruction set of RISC-V32 is unified encoding, and the encoding format is relatively fixed. The position of the opcode in the instruction is 0-6 bits. The x86 and TI C28x instruction sets are non-uniform encoding, and the encoding format varies greatly.

Instruction encoding methods can be broadly categorized into unified encoding and non-uniform encoding. In general, an instruction consists of two parts: the opcode (operation code) and the operand. The opcode is mandatory for every instruction and serves as its unique identifier. Operands, which are optional, can be further divided into destination operands and source operands. Typically, the destination operand is unique, whereas there may be multiple source operands.

In instruction sets with unified encoding, the encoding format is relatively fixed, and the opcode is usually located in a consistent position within the instruction. This consistency simplifies decoding, thereby reducing hardware complexity. In contrast, instruction sets with non-uniform encoding have varied formats where both the position and length of the opcode can differ. Although this approach increases instruction density, it complicates the decoding process.

Examples of instruction sets with unified encoding include Alpha, MIPS, and RISC-V. For instance, the RISC-V32 instruction set follows a unified encoding format, with the opcode consistently located in bits 0–6. On the other hand, instruction sets like x86 and TI C28x use non-uniform encoding, exhibiting significant variability in encoding formats. The x86 instruction set has multiple bytes of optional opcodes, and the length and position of the opcode of the C28x instruction set have multiple formats, as shown in Figure 2.

1.5. Instruction Encoding Conflict

Mixed-length instruction sets often evolve from fixed-length instruction sets. Initially, these instruction sets are typically designed with shorter fixed-length instructions, enabling efficient utilization of the encoding space. However, this design leaves limited reserved space for future extensions. To accommodate additional instructions, it becomes necessary to expand the encoding space by increasing the bit width of the instructions. For instance, the ARM Thumb instruction set initially supported only 16-bit instructions. While the Thumb instruction set provided simpler functionalities, the ARM instruction set offered more complex operations. This discrepancy required frequent mode switching between ARM and Thumb, incurring significant overhead. To address this, ARM later introduced the Thumb-2 instruction set, which combined 16-bit and 32-bit instructions to enhance functionality and reduce switching overhead [20].

Table 2 Encoding conflicting instructions.

Type	Assemble Instruction	Machine instruction
16 bit	MOVB AR6, #8bit	1101 **0110** CCCC CCCC
	MOVB AR7, #8bit	1101 **0111** CCCC CCCC
	MOVB XAR5, #8bit	1101 **0nnn** CCCC CCCC
32 bit	MAC P, loc16, *XAR7++	0101 0110 0000 0111 **1000** 0111 LLLL LLLL
	MAC P, loc16, *XAR7	0101 0110 0000 0111 **1100** 0111 LLLL LLLL

Encoding conflicts in mixed-length instruction sets can be theoretically categorized into three scenarios:

(a) Conflicts between 16-bit instructions;

(b) Conflicts between 32-bit instructions;

(c) Conflicts between 16-bit and 32-bit instructions;

As shown in Table 2, the TI C28x instruction set serves as an example. Upon practical analysis, only the first two types of conflicts occur, while the third type is mathematically impossible. This can be demonstrated by considering the following: suppose there exists a 16-bit instruction encoded as A, comprising an opcode and operands, and a 32-bit instruction encoded as BX, where B represents the high 16 bits and X represents the low 16 bits. If a conflict between A and BX occurs ($A = B$), the processor might misinterpret the 16-bit instruction A either as a complete instruction or as part of a 32-bit instruction requiring further fetching of X.

This ambiguity violates the principle of deterministic state transitions in finite state machines (FSMs), which are the foundation of modern processors. An FSM requires a unique next state for each input. If $A = B$, the processor's state transition would become indeterminate, which is logically invalid. Hence, conflicts between 16-bit and 32-bit instructions are mathematically impossible.

However, encoding conflicts within 16-bit or 32-bit instructions can occur and pose challenges in determining instruction boundaries. These conflicts, combined with the mixed-length nature of the instruction set, increase the complexity of instruction fetching and decoding, further impacting the processor's execution efficiency.

Based on the above analysis, it is difficult to determine the instruction boundary for coding conflicts of mixed-length instructions.

2. Three-Stage Decoding Algorithm and Optimization

2.1. Overall process

According to the execution tasks and the corresponding objectives, the overall process of the decoding algorithm can be divided into two main parts, as illustrated in the Figure 3. The first part involves the reading of machine instructions, represented by the left frame, which employs a two-level reading strategy to handle different instruction lengths. The second part pertains to the decoding of machine instructions, shown in the right frame, which is divided into three stages, each addressing a specific problem. A detailed introduction to these processes is provided in the following sections.

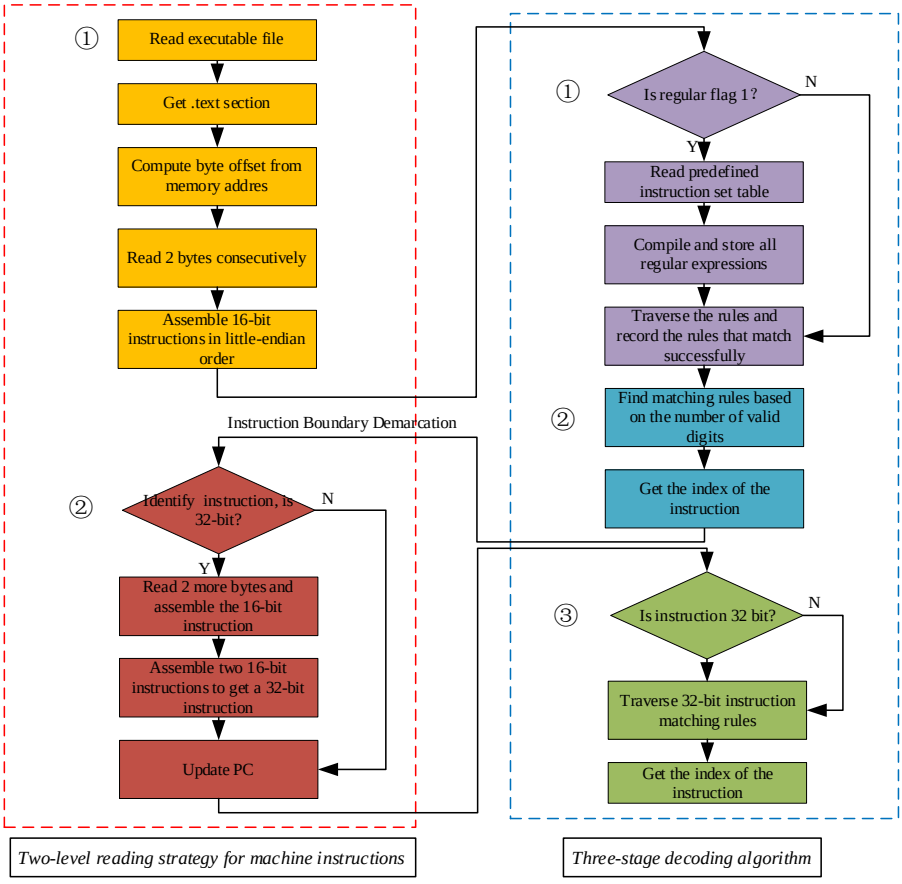


Fig 3. Fetch strategy and decoding algorithm frame

2.2. Two-level Instruction Fetch Strategy

This section introduces a two-level strategy for reading machine instruction, designed to address two main challenges:

(1) Parsing the byte stream and assembling instructions from executable files.

(2) Demarcating the boundaries in mixed-length instruction sets.

After the high-level language is translated by a compiler, it is converted into machine instructions stored in binary executable files. The decoding process requires parsing these machine instructions sequentially. For the RISC-V architecture, each address corresponds to one byte, and a 32-bit instruction occupies four consecutive addresses. If the instruction is not a branch or jump, the next step updates the program counter (PC) by adding 4 to its current value (PC = PC + 4), pointing to the next instruction. The same principle applies to 64-bit RISC-V instructions.

However, DSP chips differ in addressing methods. While RISC-V uses byte addressing, some DSP chips adopt word addressing, where the word length is typically 16 bits. This difference introduces challenges when parsing machine instructions from binary files, as the addressing mechanism impacts how instructions are read and interpreted. For example, a DSP chip with 16-bit word addressing would require decoding logic that accounts for the word boundaries, which may complicate the parsing process in a mixed-length instruction set environment.

Algorithm 1: machine instruction read strategy

```
Input: file_path, mem_addr
Output: machine instruction
// get code segment and check format of file
abfd ← bfd_openr(file_path);
if bfd_check_format(abfd) == false then
   return 0;
end
byte_data ← get_section('.text');
// compute the byte offset from mem_addr
offset ← (mem_addr – text_begin_addr) * 2;
// the first stage to read high 16bit of instruction
packet[0] ← byte_data[offset++];
packet[1] ← byte_data[offset++];
insn = packet[1] << 16 + packet[0];
decode_info ← decode(insn);
// the second stage to read low 16 bit of instruction
if decode_info.is_32bit_insn == true then
      packet[0] ← byte_data[offset++];
      packet[1] ← byte_data[offset++];
      insn_low ← packet[1] << 16 + packet[0];
      insn ← insn<<16 + insn_low;
end
return insn;
```

Parsing an executable file provides only a byte stream of machine instructions. To process these instructions, we must assemble the byte stream into complete machine instructions, considering the system's endianness. There are two common endian formats for organizing machine instructions in executable files: little endian and big endian. In little endian, the low bits of an instruction are stored at a low memory address, while the high bits are stored at a high address. In big endian, this order is reversed.

For example, as shown in Figure 3, if two consecutive bytes 84 and 92 appear in the executable file, they are assembled into the 16-bit instruction 9284 in little endian. Similarly, if four consecutive bytes a9, cd, 0f, 00 are processed,

little-endian assembly produces two 16-bit instructions, cda9 and 000f, which are then combined into the 32-bit instruction cda9000f.

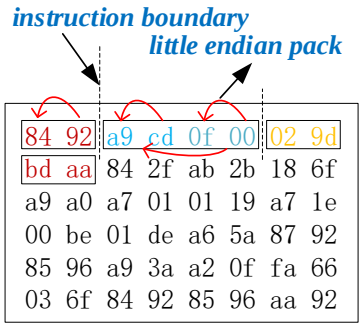


Fig 4. Little-endian pack of machine instructions

After the above stages, the 16 bits of the instruction are obtained. Since the mixed instruction set contains both 16-bit instructions and 32-bit instructions, it is impossible to determine whether the 16-bit data obtained at this time is a complete 16-bit instruction or the high 16 bits of a 32-bit instruction. Therefore, it is necessary to first determine the instruction boundary and use the 16-bit data as the input of the decoding algorithm. If there is no coding conflict, the instruction can be uniquely determined based on the 16-bit data; if there is a coding conflict, according to the analysis in section 3.5, the specific instruction cannot be determined, but the length of the instruction can be determined. After obtaining the length of the instruction, if it is a 16-bit instruction, the 16-bit data is already a complete instruction and is returned directly; if it is a 32-bit instruction, the 16-bit data is further read and assembled according to the corresponding endian sequence.

After assembling the instructions, the next challenge lies in identifying the instruction boundaries, particularly in mixed-length instruction sets that include both 16-bit and 32-bit instructions. At this stage, only 16-bit data is initially obtained, making it unclear whether it represents a complete 16-bit instruction or the high 16 bits of a 32-bit instruction. A three-stage decoding algorithm is used to address this ambiguity :

(1) Boundary Determination: The algorithm first determines whether the 16-bit data is a complete instruction or part of a longer one.

(2) Conflict Resolution: If no encoding conflict exists, the instruction is uniquely identified based on the 16-bit data. Otherwise, as discussed in Section 3.5, the algorithm identifies the instruction's length.

(3) Final Assembly: If the instruction is 16-bit, it is returned directly as the complete instruction. If it is a 32-bit instruction, additional 16-bit data is fetched and assembled according to the system's endian format.

This approach ensures accurate decoding of mixed-length instruction sets, addressing challenges posed by varying instruction lengths and encoding conflicts.

2.3. Three-Stage Decoding Algorithm

The three-stage decoding algorithm is a software decoding approach for mixed-length instruction sets. Mixed-length instruction sets consist of multiple fixed-length instructions, requiring multi-stage decoding to accurately identify each instruction. This paper uses a mixed instruction set of 16-bit and 32-bit instructions as an

example, focusing on resolving encoding conflicts through a three-stage decoding process.

Algorithm 2: two stage decode algorithm

```
Input: inst(16bit machine instruction)
Output: decode_info
// init regular expression once
If regex_init == 0 then
  for i = 0 to opcodes_table_len do
    regcomp(&re[i], opcodes[i].regex);
  end for
  regex_init ← 1;
end
// first stage to traverse all the regular rule
for i = 0 to opcodes_table_len do
  If regexec(&re[i], inst) == 0 then
    match_index[cur++] ← i;
  end
end for
op_index ← 0;
// invalid instruction and no rule can be matched
if cur < 1 do
   return NULL;
end
// second stage to determine which 16bit instruction or
whether it is a 32bit instruction
for i = 1 to cur do
    if compare(opcodes[op_index], opcodes[i]) < 0 then
       op_index ← i;
    end
end for
// third stage to determine which 32bit instruction
if is_32bit(op_index) do
    inst ← get_and_pack_16();
    op_index ← resolve_32(inst);
end
return opcodes[op_index];
```

The algorithm begins by defining matching rules for all instructions in the instruction set. A predefined table is constructed, including fields such as instruction name, matching pattern , instruction length, and associated processing function. Each rule uniquely corresponds to one instruction. Additional user-defined fields may also be included as needed.

In the first stage, a 16-bit instruction is read, and all matching rules are traversed. Successfully matched rules are recorded, but due to encoding conflicts, multiple matches may exist for a single input.

In the second stage, all successfully matched rules are sorted according to the number of valid digits of the rules. The rules with more valid digits have higher priority and are ranked in front. In fact, the matching rules with more valid digits have smaller instruction space and more accurate matching. At this time, the sorted second-layer matching rules are traversed, and the first successfully matched rule is the best matching

rule, that is, the correct mode. At this time, for this 16-bit data, it is possible to determine which 16-bit instruction it is, or whether it is the high 16 bits of a 32-bit instruction. If the 16-bit instruction is identified as a complete instruction, the decoding ends; if the 16-bit instruction is identified as the high 16 bits of a 32-bit instruction, the third stage is entered.

The third stage reads additional 16-bit data to assemble the full 32-bit instruction in little-endian order. For certain 32-bit instructions, the high 16 bits serve as opcodes, allowing the instruction to be uniquely identified. However, for instructions with high 16-bit encoding conflicts, further decoding is required to resolve ambiguities. The process for resolving these conflicts is similar to the handling of 16-bit instruction conflicts and will not be repeated here.

Overall, the algorithm operates as follows: the first stage filters out matching patterns; the second stage prioritizes matches by the number of valid bits, selecting the best match; the third stage addresses encoding conflicts for 32-bit instructions. This approach ensures accurate decoding of mixed-length instruction sets despite the presence of encoding conflicts.

2.4. Priority-Based Optimization

In the above algorithm, for each instruction with coding conflicts, the second stage requires looping through all matching rules to determine the best match. This process is time-consuming and has room for optimization. By analyzing the instruction set in advance, all instructions can be sorted lexicographically. Using a sliding window algorithm, the largest window of instructions with common prefixes can be identified, representing a group of conflicting instructions. Within this window, instructions are further sorted by the number of exact matching bits. Instructions with more exact matching bits correspond to smaller instruction spaces and more accurate matches, as detailed in Algorithm 3.

This optimization moves the sorting process from the second stage at runtime to the framework initialization phase, where it only needs to be executed once. For architectures with minimal instruction set changes, the instruction set table can be directly optimized during its definition. In such cases, the sorting process is entirely moved from runtime to the definition phase, eliminating the need for initialization sorting.

From the perspective of compiler optimization, this approach utilizes loop expansion to shift the sorting loop in the two-stage decoding algorithm from runtime to earlier stages. The process is reduced from multiple executions during runtime to a single execution before the program runs. This not only improves efficiency but also ensures the optimized instruction set table is ready for immediate use during decoding.

Algorithm 3:optimized two stage decode algorithm

```
Input: machine_instruction
Output: decode_info
// first stage to traverse all the regular rule
If optimized == 0 then
  sort(opcodes);
  // sliding window to sort the opcodes again
  left,right=0;
  while right < opcodes_table_len do
    if left = right or has_common_prefix(opcodes[left,
          right]) then
      right++;
    else
```

```
        sort(opcodes, left , right);
        left++;
      end
    end while
    optimized←1;
end
// init regular expression once
If regex_init == 0 then
    for i = 0 to opcodes_table_len do
      regcomp(&re[i], opcodes[i].regex);
    end for
    regex_init ← 1;
end
// traverse all the match rule, and the first rule is the best
rule
for i = 0 to opcodes_table_len do
    If regexec(&re[i], inst) == 0 then
      match_index[cur++] ← i;
      break;
    end;
end for
return opcodes[op_index];
```

3. Evaluation

This paper establishes an experimental platform running Ubuntu 22.04.4, equipped with an Intel 9400f 2.90GHz CPU, 16GB of memory, and a 512GB hard disk. The test and verification data set is based on TI's C28x instruction set, compiled using CCS 12.0 to generate executable files.

The C28x instruction set includes approximately 426 instructions (excluding compatibility with the C27x set), comprising around 299 16-bit instructions and 127 32-bit instructions. For 16-bit instructions, the majority use the upper 8 bits for opcode encoding and the lower 8 bits for operand encoding. A subset uses all 16 bits for opcode encoding without requiring operand encoding, such as status register setup instructions. Another subset uses the upper 12 bits for opcode encoding and the lower 4 bits for operand encoding, such as shift instructions. A very small number of instructions employ other bit patterns for encoding.

For the 32-bit instruction set, most instructions use the upper 22 bits for opcode encoding and the lower 8 bits for operand encoding. A small number of instructions adopt different bit lengths for encoding. This encoding approach efficiently utilizes the instruction set's encoding space while ensuring scalability for future extensions.

3.1. Correctness Testing

3.1.1. Automated Testing Framework

Disassembly involves the direct translation of machine instructions into assembly instructions without introducing additional logic, making it an ideal test scenario for evaluating the machine instruction decoding algorithm presented in this paper.

As part of the testing process, the DFT algorithm is written in assembly language and compiled into an ELF-format executable file targeting the C28x architecture. Using TI's disassembler dis2000, the executable file is processed to obtain the standard output, which includes the disassembly results alongside additional redundant information, such as symbol tables and metadata. This redundant information is filtered to focus solely on the relevant disassembly output.

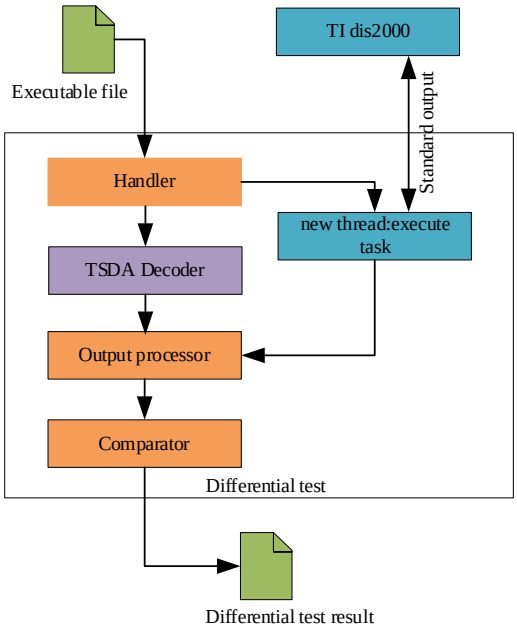


Fig 5. Automated testing framework

The overall process is as figure 5 shows: the handler receives the path of the executable file, processes it, and creates a separate thread to launch the dis2000 process for communication. The input is the entire executable file, and the output is the disassembly result, including symbol information. Unnecessary spaces and commas in each disassembled line are removed, focusing solely on opcodes, source operands, and destination operands. These strings are tokenized, with each line returning an array of string tokens. This process is akin to lexical analysis in compilers.

The comparator compares strings line by line, recording any differences for further persistent output. A synchronization barrier is set in the comparator to ensure that both the main and child threads complete their tasks and that all standard outputs are captured.

3.1.2. Single-instruction testing

Single instruction testing aims to ensure comprehensive coverage of the instruction set and verify the decoding accuracy of each instruction. Writing test cases in high-level languages often relies on the compiler to translate these cases into machine instructions. However, the compiler's instruction selection and optimization processes may prevent certain instructions from being generated, resulting in incomplete testing.

To address this issue, test cases should be written directly in assembly language, with each instruction manually specified. This approach bypasses the compiler's optimization and ensures that all instructions are tested individually and directly translated into the corresponding machine instructions.

For example, as shown in Table 3, the automated testing framework outputs results for all cases, but only the failed cases are highlighted and retained for further analysis, while passed cases are ignored. This selective focus enhances testing efficiency and aids in debugging.

Table 3 Single instruction test results.

Machine instruction	Standard output	TSRDecoder output	Pass
0xff56	ABS ACC	ABS ACC	true
0x28035555	MOV @0x3, #0x5555	MOV @0x3, #0x5555	true

0x6147	SB 71, EQ	SB 71, NEQ	false

3.2. Performance Evaluation

To better evaluate the performance of the algorithm and its optimizations, two types of datasets were used in this study: custom algorithms and standard libraries. The custom dataset includes DFT and FFT assembly implementations developed specifically for this evaluation. The standard library dataset was selected from widely used DSP libraries, leveraging examples from the TI C2000 toolkit, such as FixedPoint_CFFT, FixedPoint_RFFT, FixedPoint_FIR32, and FixedPoint_IIR32.

The performance of TSDA and OTSDA, as described in Sections 4.3 and 4.4, was evaluated using six test datasets. The execution times for each algorithm are presented in Table 4. The results indicate that OTSDA achieves significant optimization, with efficiency improvements ranging from 70% to 73%. These findings highlight the effectiveness of the proposed optimization approach, with OTSDA consistently outperforming TSDA across all datasets.

Table 4 Single instruction test results.

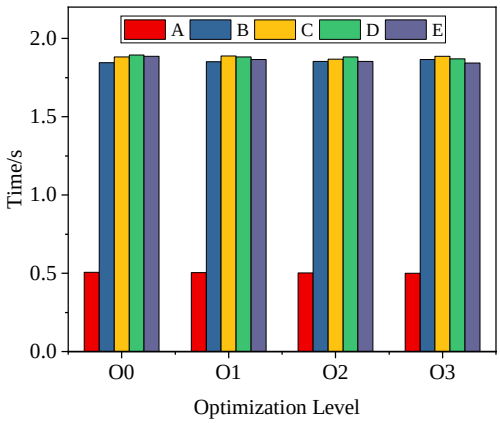
Data set	TSDA execution time/s	OTSDA execution time/s	Proportion
DFT	1.845	0.505	0.274
FFT	1.898	0.512	0.270
CFFT lib	48.080	14.116	0.294
RFFT lib	47.064	12.698	0.270
FIR32 lib	43.015	11.778	0.274
IIR32 lib	43.856	11.963	0.273

In this study, several optimization schemes were designed to compare and analyze whether the optimization effect of OTSDA is similar to that of loop unrolling:

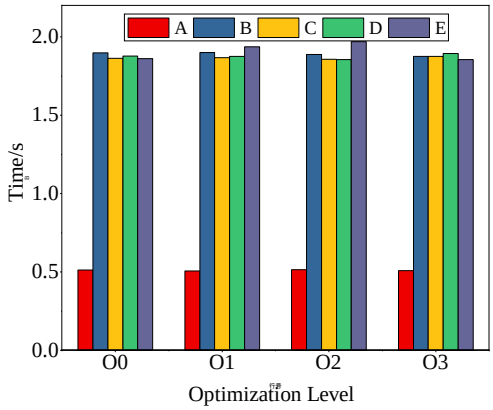
- Scheme A: Optimized three-stage decoding algorithm (OTSDA)
- Scheme B: Original three-stage decoding algorithm (no loop unrolling)
- Scheme C: Loop unrolling applied to the second stage with a factor of 2
- Scheme D: Loop unrolling applied to the second stage with a factor of 3
- Scheme E: Loop unrolling applied to the second stage with a factor of 4

Additionally, the GCC compiler was configured to use various optimization levels (O0, O1, O2, O3) to examine their impact on the performance of TSDA and OTSDA.

The performance of the decoding algorithm was tested using custom DFT and FFT assembly algorithms. The results, as shown in the figure 6, indicate that although TSDA benefits from loop unrolling, the optimization effect is minimal. In contrast, OTSDA consistently achieved the best performance across all optimization levels.



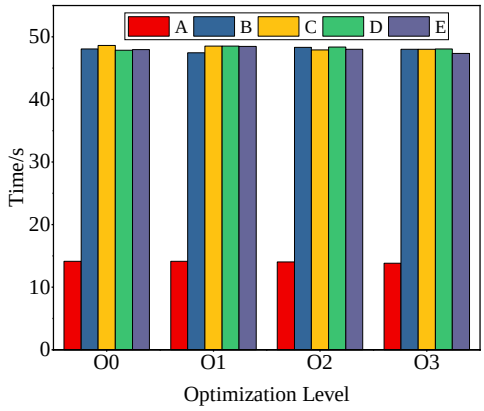
(a) DFT



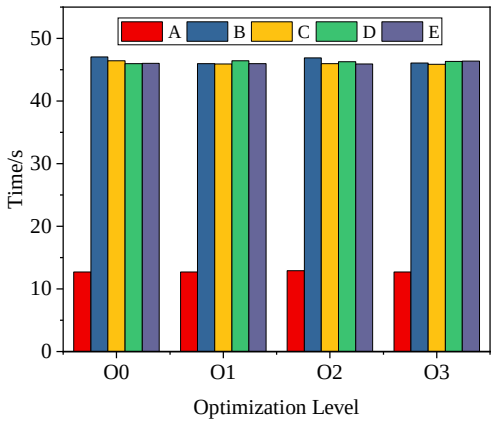
(b) FFT

Fig 6. Performance testing of different optimization schemes on user code

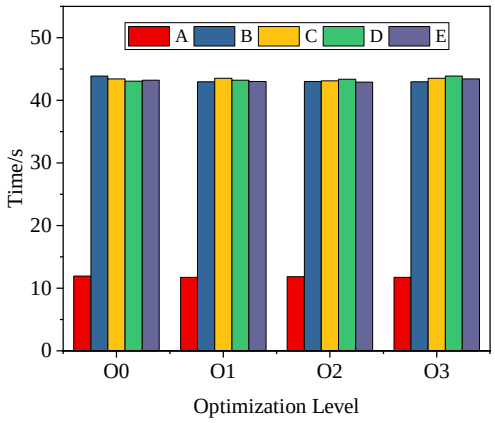
The decoding algorithm's performance on the standard libraries CFFT, RFFT, FIR32, and IIR32 is shown in Figure 7. Consistent with the previous results, loop unrolling had minimal impact on optimization, while OTSDA demonstrated the most significant performance improvement.



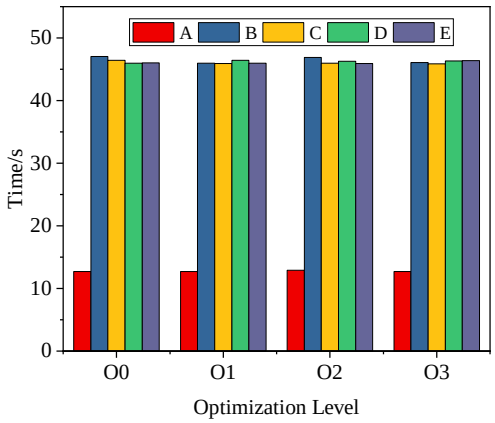
(a) CFFT



(b) RFFT



(c) FIR32



(d) IIR32

Fig 7. Performance testing of different optimization schemes on the standard library

Based on the test results, it is evident that the TSDA loop unrolling scheme did not yield noticeable optimization. This is likely due to the presence of numerous branch jump instructions within the loop. Loop unrolling primarily improves performance by increasing instruction-level parallelism and reducing the overhead of loop branch jumps. However, it does not eliminate branch jump instructions in this case, resulting in negligible performance gains.

In contrast, OTSDA achieves superior optimization by addressing upper-level logic and the second stage of the decoding algorithm. It transforms multiple runtime executions into a single pre-runtime execution, thereby avoiding repeated processes in the second stage. This

approach significantly reduces branch jump instructions and enhances pipeline performance, leading to a noticeable improvement in decoding efficiency.

4. Conclusion

This paper proposes a machine instruction reading strategy and a decoding algorithm for mixed-length instruction sets with non-uniform encoding. The reading strategy directly processes executable files in two stages: the first stage handles the assembly of 16-bit instructions, and the second stage processes 32-bit instructions. The decoding algorithm is divided into three stages: the first stage addresses 16-bit encoding conflicts, the second stage resolves 16-bit instruction conflicts, and the third stage solves 32-bit instruction conflicts.

Furthermore, the algorithm is easy to implement compared to traditional decoder generation methods, which often require learning a specific description language. In contrast, our method only needs the definition of the instruction set table. An optimization is also proposed to improve the algorithm's performance: the loop in the second stage is moved before program execution, or during the instruction set table definition stage.

The algorithm was tested on both user code and standard library functions, achieving significant results. The maximum efficiency improvement was around 70%, demonstrating good generalization ability.

Declaration of Competing Interest

The authors declare no conflict of interest.

Reference

1. N. Jay and B. P. Miller, “Structured random differential testing of instruction decoders,” in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 84–94.
2. Hsieh, Wilson C. et al. “Reverse-Engineering Instruction Encodings.” *USENIX ATC, General Track* (2001).
3. A. Akram and L. Sawalha, "A Survey of Computer Architecture Simulation Techniques and Tools," in IEEE Access, vol. 7, pp. 78120-78145,2019, doi:10.1109/ACCESS.2019.2917698.
4. G. Wang, Z. Zhu, S. Li, X. Cheng and D. Meng, "Differential Testing of x86 Instruction Decoders with Instruction Operand Inferring Algorithm," 2021 IEEE 39th International Conference on Computer Design (ICCD), Storrs, CT, USA, 2021, pp. 196-203, doi: 10.1109/ICCD53106.2021.00040.
5. T. E. Jeremiassen, "Sleipnir. An instruction-level simulator generator," *Proceedings 2000 International Conference on Computer Design*, Austin, TX, USA, 2000, pp. 23-31, doi: 10.1109/ICCD.2000.878265.
6. [6] N. Fournel, L. Michel and F. Pétrot, "Automated generation of efficient instruction decoders for Instruction Set Simulators," 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 2013, pp. 739-746, doi:10.1109/ICCAD.2013.6691197.
7. Free Software Foundation, Inc.,“GNU binutils,” http://www.gnu.org/software/binutils.
8. Free Software Foundation, Inc., “GDB: The GNU Project Debugger,” https://www.sourceware.org/gdb/
9. F. Bellard, “QEMU. Open Source Processor Emulator,” wiki.qemu.org/Main Page, version 6.2.0.
10. Meyerowitz T, Sprinkle J, Sangiovanni-Vincentelli A. A visual language for describing instruction sets and generating decoders[C]//Proc. of the 4th ACM OOPSLA Workshop on Domain Specific Modeling, Vancouver, BC. 2004: 23-32.
11. H. Theiling, “Generating decision trees for decoding binaries,” in Proceedings of the ACM SIGPLAN Workshop on Languages,

Compilers and Tools for Embedded Systems (LCTES), jun 2001, pp. 112–120.
12. Tadros, Lillian. “Functional Analysis And Performance Evaluation Of Decoder Decision Tree Generation Algorithms.” *European Conference on Modelling and Simulation* (2023).
13. W. Woodruff, N. Carroll and S. Peters, "Differential analysis of x86-64 instruction decoders," 2021 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, 2021, pp. 152-161, doi: 10.1109/SPW53761. 2021.00029.
14. G. Wang, Z. Zhu, X. Cheng and D. Meng, "In-depth Testing of x86 Instruction Disassemblers with Feedback Controlled DFS Algorithm," 2022 IEEE 40th International Conference on Computer Design (ICCD), Olympic Valley, CA, USA, 2022, pp. 463-470, doi:10.1109/ ICCD563 17.2022.00075.
15. Sarma B, Dasgupta S. Study and Analysis of ELF Vulnerabilities in Linux[J]. International Journal of Innovative Research in Engineering & Management(IJIREM) ISSN: 2350-0557, Volume-1, Issue-3, November-2014.
16. Galuzzi C, Bertels K. The instruction-set extension problem: A survey[J]. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2011, 4(2): 1-28.
17. Intel, Intel 64 and IA-32 Architectures Software Developer's Manual, https://www. intel.com/content/www/us/en/developer/articl es/technical/intel-sdm.html#combined#comb ined.
18. RISC-V Foundation, The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213, Dec. 2019. https://riscv.org/wp-content/uploads/2019/12/ riscv-spec-20191213.pdf.
19. MIPS Technologies, MIPS32 Architecture for Programmers Volume I: Introduction to the MIPS32 Architecture, Document Version 6.06, Dec. 2012. https://s3-eu-west-1.amazon aws.com/downloads-mips/documents/MD000 86- 2B-MIPS32BIS-AFP-6.06.pdf.
20. Arm Limited, Arm Architecture Reference Manual for A-profile Architecture, Version L.a, 2023. https://developer.arm.com/docu mentation/ddi0487/latest/.