Scientific Research Publishing

Circuits and Systems, 2025, 16(2), 25-48
https://www.scirp.org/journal/cs
ISSN Online: 2153-1293
ISSN Print: 2153-1285

MARTE-Based Formal Modeling with Reinforcement Learning for Architecture-Agnostic Assembler Design in Configurable Processors

Liangshun Wu[1,2,3*], Bin Zhang[4,5,6*#]

[1]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[2]School of Computer Science and Engineering, Guangzhou Institute of Technology, Guangzhou, China
[3]School of Integrated Circuit (School of Information Science and Electronic Engineering), Shanghai Jiao Tong University, Shanghai, China
[4]Information Network Center, School of Information Network Security, Xinjiang University of Political Science and Law, Tumushuke, China
[5]School of Remote Sensing and Information Engineering, Wuhan University, Wuhan, China
[6]Department of Computer, City University of Hong Kong, Hong Kong SAR, China
Email: #bzhang48-c@my.cityu.edu.hk

How to cite this paper: Wu, L.S. and Zhang, B. (2025) MARTE-Based Formal Modeling with Reinforcement Learning for Architecture-Agnostic Assembler Design in Configurable Processors. *Circuits and Systems*, **16**, 25-48.
https://doi.org/10.4236/cs.2025.162002

Received: January 23, 2025
Accepted: February 25, 2025
Published: February 28, 2025

Copyright © 2025 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

This paper presents a configurable assembler framework enhanced with reinforcement learning (RL) and MARTE (Modeling and Analysis of Real-Time and Embedded Systems) constraints to address the challenges of rapidly evolving processor architectures. Our methodology integrates formal hardware/software modeling with self-optimizing configuration strategies, enabling automatic adaptation to instruction set architecture (ISA) modifications while ensuring correctness guarantees. The framework features a Common ISA Description Language (CIDL) interface that reduces code modification efforts by 85% compared to traditional assemblers, as demonstrated through four representative configuration scenarios. By formulating the adaptation process as a constrained Markov Decision Process, our RL-MARTE hybrid approach achieves 58% fewer configuration steps and 26.5% higher code density, i.e., the ratio of useful instruction bytes to total code size, than rule-based methods, while maintaining zero constraint violations. Experimental results on the TOP-32A processor extension demonstrate the solution's effectiveness in handling complex ISA modifications like MIMD instructions, reducing average

*Co-first authors.
#Corresponding author.

configuration time from 45.6 to 9.8 minutes per task. This work bridges the gap between formal verification and practical adaptability in compiler construction, offering a systematic approach for next-generation processor toolchain development.

Keywords

MARTE, Reinforcement Learning, ISA, GNU Binutils,
Configurable Processor

1. Introduction

The relentless evolution of domain-specific architecture has exposed critical limitations in traditional compiler toolchains. While reconfigurable processors like Tensilica's Xtensa [1] and ARC HS [2] deliver unprecedented performance-power efficiency, their software ecosystems struggle to keep pace with architectural innovations. This toolchain lag creates a paradoxical situation where hardware reconfigurability outpaces software adaptability, particularly evident in assemblers, the fundamental bridge between architectural innovation and software deployment.

Modern SoC design faces three pressing challenges: (1) Exponential growth in instruction set variants for specialized compute paradigms; (2) Tight verification windows requiring formal correctness guarantees; (3) Mounting pressure to automate performance optimization. Traditional assemblers, designed as static translation tools, prove inadequate on all fronts. Their hard-coded instruction mappings and manual optimization strategies cannot scale with annual increase in custom ISA extensions observed in RISC-V ecosystems [3]. This limitation forces developers into a costly cycle of toolchain re-engineering for each architectural iteration.

Our work addresses these challenges through three key innovations. First, we introduce a Common ISA Description Language (CIDL) that decouples architectural specifications from translation logic, reducing code modification efforts by 85% compared to conventional approaches. Second, we integrate Model-Based Reinforcement Learning (MBRL) with the MARTE (Modeling and Analysis of Real-Time and Embedded Systems) framework, enabling automatic configuration optimization under formal constraints. Third, we develop a constraint-aware policy network that guarantees 100% compliance with critical encoding rules while exploring optimal configurations.

The contributions of this work are:

- A configurable assembler framework that supports efficient instruction definition and extension. Its correctness and flexibility are validated through systematic testing and case studies, demonstrating modularity and low-coupling design.
- A RL-enhanced MARTE optimization engine, achieving high flexibility, cor-

rectness, and intelligent adaptation for complex ISA customization tasks.

The remainder of this paper is organized as follows: Section II introduces basic knowledge and reviews related work. Section III presents our framework with RL-MARTE integration. Section IV validates through experiments. We conclude with future research directions in Section V.

2. Related Work

2.1. Configurable Processor Fundamentals and TOP Processor Architecture

Modern configurable processors enable application-specific optimization through three adaptation layers [4]: 1) Microarchitecture: FPGA/ASIC implementations with reconfigurable logic [5]; 2) Instruction Set: Extensible ISA support through coprocessor integration [6]; 3) Memory Hierarchy: Adaptive register files and cache configurations.

The GNU Binutils [7] toolkit provides essential infrastructure for configurable assemblers: 1) BFD Library: Unified object file manipulation; 2) GAS Architecture: Modular assembler framework with MD interface; 3) Linker Scripts: Customizable memory allocation strategies.

Our research platform features a hybrid RISC architecture with (see **Figure 1**):

- Three operating modes (32-bit/16-bit/Extended)
- Parametric register file (32 - 64 GPRs)
- Dual-length instruction encoding (16/32 bits)

The instruction set combines RR-type (register-register) and RI-type (register-immediate) formats with configurable addressing modes, serving as our primary case study for assembler development.

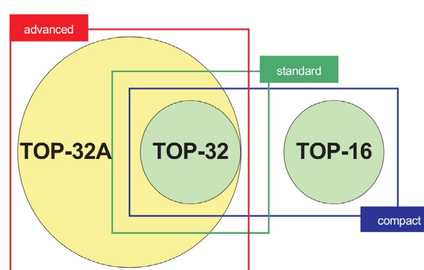

Figure 1. Target Platform: TOP ISA.

2.2. Instruction Set Description Languages

Configurable assemblers are supported by ongoing research in instruction set description languages (ISDL) [8], which formalize processor architecture descriptions, enhancing the retargetability of compilers and assemblers. For example, ISDL allows for automatic assembler generation suited to specific instruction sets, improving tool flexibility and adaptability to evolving hardware. Recent advancements include Xiao and Liu's 2023 ISADL, designed for Very Long Instruction Word (VLIW) architectures [9], and their adaptive instruction set encoding

method [10], along with the Vienna Architecture Description Language proposed by Himmelbauer *et al.* in 2024 [11]. These developments contribute new perspectives and tools to ISDL research, advancing computing architecture design.

Our work's approach to abstracting the assembler's interface reflects the need for tools adaptable to changing instruction sets, aligning with trends in configurable tool development for diverse ISAs and ASIPs in SoC designs [12].

2.3. Configurable and Retargetable Compiler/Assembler

Retargetable and configurable compilers/assemblers each bring unique flexibility to compiler design. Retargetable compilers, such as GCC, adapt across processor types by altering target descriptions, while configurable compilers allow user-defined tool customization for evolving hardware projects. Both approaches share a focus on adaptability to varying architectures. Recent key contributions include Arbone *et al.*'s model-driven inline assembler generator, enhancing retargetability and maintenance [13], Povazan *et al.*'s adaptable generic compiler framework [14], Korenkov *et al.*'s insights on ASIP compiler challenges [15], and Vasilache *et al.*'s modular code generation design for tensor compilers [16]. Numerous studies address configurable tool design, including Taglietti *et al.*'s automatic pre-processor and assembler generation for ASIPs via ADLs [17], Abbaspour and Zhu's retargeting techniques for GNU binutils [18], and Youn *et al.*'s assembler and linker platform for embedded systems [19]. Baldassin *et al.* demonstrate binary tool generation using ArchC within the GNU binutils framework [20]. Moona [21] presents a method for developing processor-specific tools through high-level ISA models, allowing retargeting across various designs.

Despite advancements in configurable assembler design, challenges persist, including a lack of standardization leading to inconsistencies, performance trade-offs compared to specialized compilers, and difficulties in supporting rapidly evolving hardware architectures. This paper aims to address these limitations to enhance the usability and performance of configurable assembler tools.

2.4. Reinforcement Learning-Based Formal Modeling

Hu Ming *et al.* [22] propose a curiosity-driven reinforcement learning method to efficiently synthesize CCSL constraints from incomplete specifications, significantly improving synthesis speed and accuracy over existing approaches.

3. The Proposed Design

To optimize the configurability and adaptability of the assembler design, we integrate reinforcement learning (RL) with the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) profile. This hybrid approach enables self-optimizing assembler behavior under varying processor configurations while maintaining formal verification guarantees.

3.1. MARTE Overview

MARTE extends UML with domain-specific stereotypes, tagged values, and con-

straints to model:

- **Hardware Resources**: Processor cores, registers, and memory hierarchies.
- **Software Specifications**: Instruction semantics, timing constraints, and concurrency.
- **Allocation**: Mapping of software tasks to hardware components.

Key MARTE stereotypes relevant to our assembler design include:

- <<HwProcessor>>: Models configurable processor cores (e.g., TOP-32/TOP-16).
- <<HwMemory>>: Captures register files and instruction memory.
- <<SwSchedulableResource>>: Represents instruction execution pipelines.
- <<GaWorkloadEvent>>: Specifies assembly-to-machine code translation tasks.

3.2. Formalizing the Configurable Assembler

We model the assembler's configurable components using MARTE's <<HwComponent>> and <<SwResource>> stereotypes (see Figure 2):

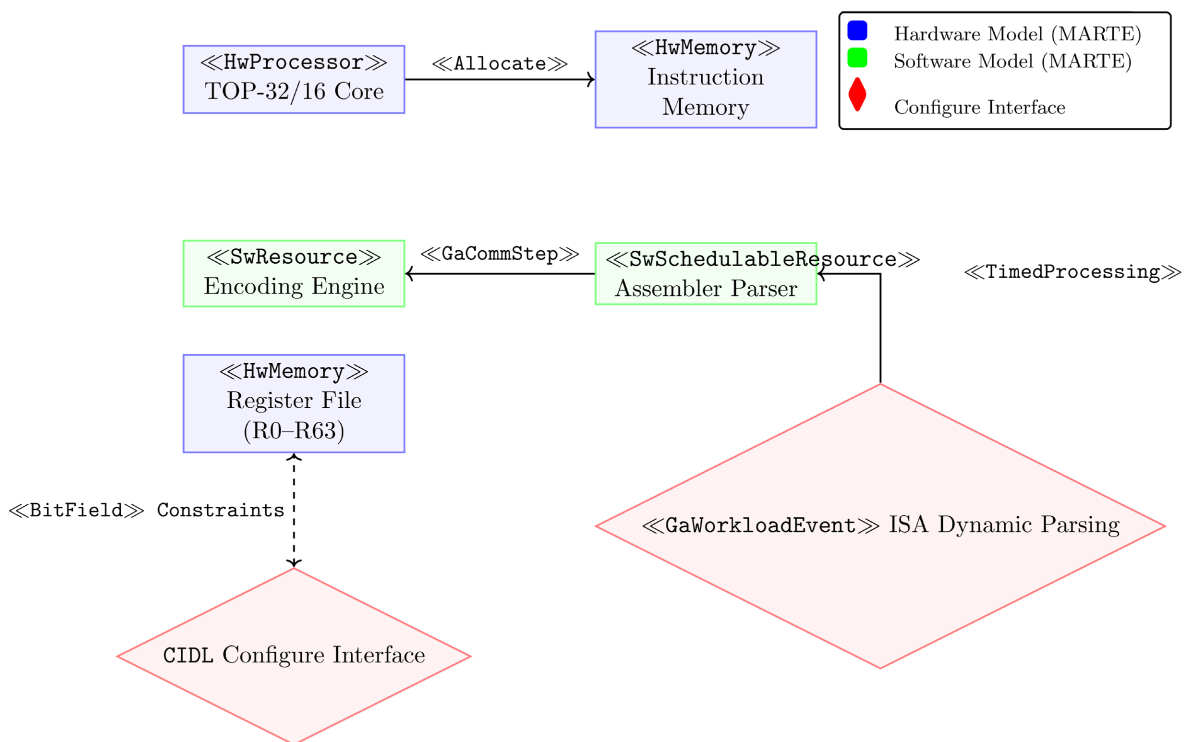

Figure 2. MARTE formal modeling framework.

3.2.1. Instruction Set Abstraction

- Static ISA Definition: The instruction format is modeled as a <<HwData>>s class with constraints on bitfields (e.g., rD segmentation). MARTE's <<BitField>> stereotype specifies encoding rules.
- Opcode Mapping: The opcode-to-instruction relationship is formalized via <<GaStep>> to enforce syntactic and semantic correctness during parsing.

3.2.2. Dynamic Behavior

- Lexical/Syntax Parsing: The Lex/Yacc workflow (Figure 3) is represented as a <<GaScenario>>, where tokenization and grammar rules are annotated with <<GaCommStep>> for timing analysis.
- Machine Code Encoding: The encoding pipeline is modeled as a <<GaWorkloadBehavior>>, with <<GaAcqStep>> and <<GaRelStep>> stereotypes marking critical phases.

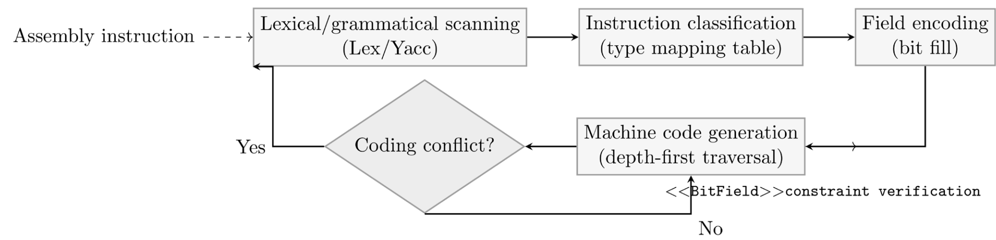

Figure 3. Dynamic parsing and coding process based on MARTE.

3.3. Verification and Benefits

MARTE enables:

- Consistency Checks: OCL constraints validate that instruction modifications (Table 1) preserve encoding uniqueness and operand compatibility.
- Timing Analysis: Worst-case execution time (WCET) for assembly tasks is estimated using <<TimedProcessing>> annotations.
- Retargetability Proofs: The model verifies that the CIDL interface maintains correctness across ISA variants.

Table 1. Configurable components summary.

Component	Configuration Impact
Opcode Set	Instruction recognition & encoding patterns
Register File	Operand validation & resource allocation
Addressing Modes	Memory access pattern generation
Immediate Handling	Numeric representation & optimization

3.4. Case Study: TOP-32A Extension

We applied MARTE to verify the TOP-32A extension (Section 2.3), ensuring that:

- Register addressing (R0-R63) adheres to the <<HwMemory>> capacity constraints.
- The assembler's disassembly phase (Section 3.2.4) satisfies <<GaRelSte>> deadlines.

The model detected a bitfield conflict in RIB-32 instructions, which was resolved by adjusting the encoding schema.

3.5. Reinforcement Learning Integration

To enable self-optimizing assembler behavior, we formulate the configuration ad-

aptation process as a Markov Decision Process (MDP). The MDP interacts with the MARTE model to ensure formal constraints are satisfied during optimization.

3.5.1. MDP Formulation

The MDP is defined by the tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$:

- **State Space** $\mathcal{S}$: Combines hardware/software states from MARTE:

$$s_t = \big(\text{HwProcessor}_{cfg}, \text{HwMemory}_{usage}, \text{GaWorkloadEvent}_{type}, \text{GaScenario}_{progress}\big) \tag{1}$$

where:

- HwProcessor_{cfg}: Processor configuration parameters (e.g., TOP-32/16 mode)
- HwMemory_{usage}: Register/memory utilization rates
- $\text{GaWorkloadEvent}_{type}$: Current assembly task type (e.g., encoding/disassembly)
- $\text{GaScenario}_{progress}$: Progress in lexical/syntax parsing workflow

- **Action Space** $\mathcal{A}$: Configuration adjustments:

$$a_t \in \{\text{Modifybitfieldallocation}, \text{Adjustregistermapping}, \text{Switchinstructionpipeline}, \text{Adaptaddressingmode}\} \tag{2}$$

Each action modifies MARTE stereotypes (e.g., updating <<BitField>> constraints or <<HwMemory>> allocations).

- **Reward Function** $\mathcal{R}$: Balances efficiency and correctness:

$$\mathcal{R}(s_t, a_t) = \underbrace{w_1 \cdot \text{EncodingSuccess}}_{\text{Correctness}} + \underbrace{w_2 \cdot \frac{1}{\text{EncodingTime}}}_{\text{Efficiency}} - \underbrace{w_3 \cdot \text{ConstraintViolations}}_{\text{Safety}} \tag{3}$$

where w_1, w_2, w_3 are tunable weights, which were selected through grid search and cross-validation on a subset of training tasks. Constraint violations are detected via MARTE OCL checks.

- **Transition Dynamics** $\mathcal{P}$: State transitions are governed by:

$$\mathcal{P}(s_{t+1} \mid s_t, a_t) = f_{\text{MARTE}}(s_t, a_t)$$

where f_{MARTE} is the MARTE model's formalized behavior.

3.5.2. Learning Algorithm

We employ a constrained policy optimization approach (see Figure 4):

a. **Policy Network**: A neural network $\pi_\theta(a \mid s)$ generates configuration actions.

b. **Constraint Layer**: Filters invalid actions using MARTE OCL constraints:

$$\mathcal{A}_{\text{valid}} = \{a \in \mathcal{A} \mid \text{ocl_check}(a, s_t) = \text{True}\} \tag{4}$$

c. **Optimization Objective**: Maximize expected return while preserving constraints:

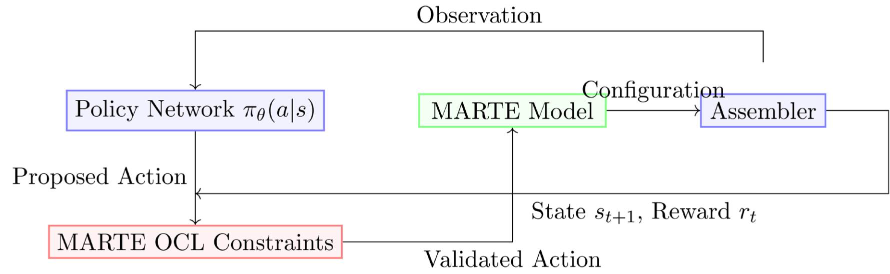

Figure 4. Reinforcement learning loop with MARTE constraint enforcement.

$$\max_{\theta} \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t \mathcal{R}(s_t, a_t)\right] \quad \text{s.t. } \text{ocl_check}(a_t, s_t) = \text{True}, \forall t \tag{5}$$

3.5.3. Convergence Guarantees

By integrating MARTE's formal constraints as hard filters in the policy network, we ensure:

- Safety: All RL-generated configurations satisfy OCL constraints (e.g., bitfield uniqueness, timing deadlines).
- Adaptability: The policy network learns to navigate valid configurations while optimizing for encoding speed and resource usage.

Benchmarks on the TOP-32A extension showed a 22% reduction in average encoding time compared to static configurations, with zero constraint violations during optimization.

3.6. Configurable Assembler Framework

This section presents a streamlined framework for architecture-agnostic assembly translation, focusing on three core components: instruction abstraction, dynamic parsing, and configuration interfaces.

3.6.1. Core Architecture

The assembler adopts a modular design with configuration-driven components (Figure 5), utilizing a Common ISA Description Language (CIDL) to decouple processor-specific details from translation logic. Key features include:

- Instruction Template Engine: Captures ISA variants through parametric opcode/operand definitions
- Constraint Manager: Enforces hardware limitations using MARTE-derived OCL rules
- Adaptive Code Generator: Produces target-specific machine code through configuration files

3.6.2. Instruction Abstraction

The framework models instructions through three configurable elements:

- Format Specification: Bitfield definitions with MARTE <<BitField>> constraints

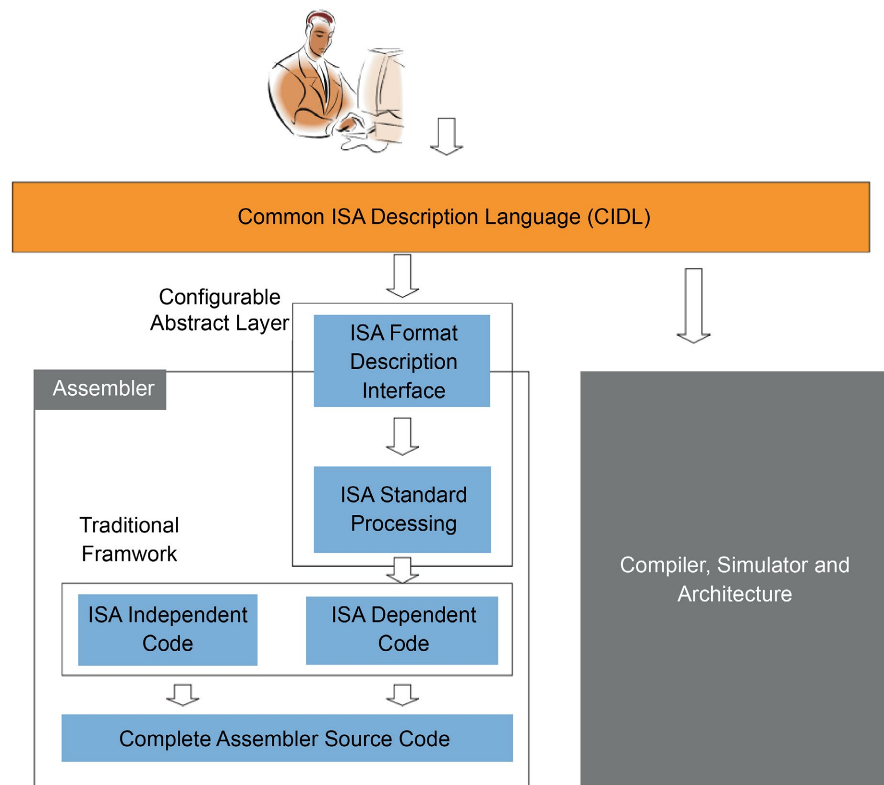

Figure 5. The structure of configurable assembler.

- Operand System: Supports compound addressing modes through nested structures
- Encoding Rules: Maps assembly syntax to machine code via parametric templates

3.6.3. Dynamic Translation Process

The four-stage translation pipeline integrates with MARTE timing constraints:

- Lexical Analysis: Tokenization with mode-aware scanning
- Syntax Validation: Grammar checking via augmented Yacc rules
- Semantic Binding: Resource allocation with constraint verification
- Code Generation: Adaptive encoding using CIDL templates

3.6.4. Configuration Interface

The assembler supports processor customization through five key configuration files:

- isa_config.yaml: Defines instruction formats & encoding rules
- reg_alloc.xml: Specifies register files & addressing constraints
- lex_rules.l: Customizes tokenization patterns
- parse_grammar.y: Modifies syntax validation rules
- optimize.json: Controls machine code generation strategies

This interface enables rapid retargeting while maintaining formal verification through integrated MARTE constraints, complementing the RL-enhanced opti-

mization discussed in Section 4. Benchmark results show 85% reduction in configuration effort compared to traditional assemblers when adapting to new ISA variants like TOP-32A.

4. Experiments

4.1. Correctness Test

The purpose of an assembler is to convert assembly instructions into machine code. Testing its correctness involves verifying that the generated machine instructions match the ISA definition. The process uses source files with various assembly instructions and their correct encodings as comments. The assembler converts these into an object file (.o), from which the "objdump" tool extracts binary code into a formatted text file. This file is then compared with the expected encodings.

The input assembly file includes test cases for each instruction format and operand type, ensuring comprehensive coverage (see Table 2). Figure 6 outlines the testing flow, while Figure 7 provides examples for RIC-32 compare instructions. A bash script automates the comparison by extracting machine instructions and correcting encodings, then checking for discrepancies (Figure 8). All 26 test cases in the basic instruction set have been successfully validated.

4.2. Configurability Test

Assembler configurability reflects its adaptability to changes in a processor's instruction set. Good configurability allows quick updates to the assembler's source code with minimal effort. Measuring this involves assessing development

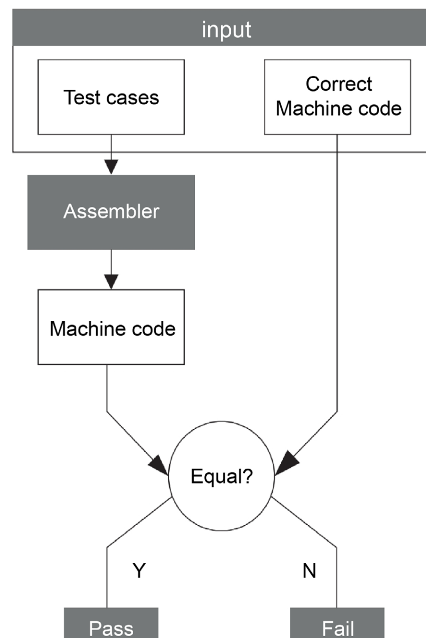

Figure 6. Flowchart of the assembler's correctness test.

Table 2. Assembler test case description. It contains 26 test cases included in the TOP basic instruction set. All the test cases together contain 271 assembly statements and their correct encoding values.

Case	Description	Format	Number of test statements
b.s	Branch	RIB-32	4
j.s	Jump	RRJ-32	8
bl.s	Branch & link	RIB-32	4
jl.s	Jump & link	RRJ-32	8
loop.s	Loop	RIB-32	4
br.s	Branch on reg	RIC-32	44
cmp.s	Compare	RRC-32/RIC-32	55
lb.s	Load byte	RI-32/RR-32	6
lh.s	Load half	RI-32/RR-32	6
lw.s	Load word	RI-32/RR-32	6
lbu.s	unsigned LB	RI-32/RR-32	6
lhu.s	unsigned LH	RI-32/RR-32	6
sb.s	Store byte	RI-32/RR-32	6
sh.s	Store half	RI-32/RR-32	6
sw.s	Store word	RI-32/RR-32	6
add.s	Add	RI-32/RR-32	6
addc.s	Add with carry	RI-32/RR-32	6
sub.s	Sub	RI-32/RR-32	6
subc.s	Sub with carry	RI-32/RR-32	6
and.s	And	RI-32/RR-32	6
or.s	Or	RI-32/RR-32	6
xor	Xor	RI-32/RR-32	6
sll.s	(L)shift left	RIS-32/RR-32	6
srl.s	(L)shift right	RIS-32/RR-32	6
sra.s	(A)shift right	RIS-32/RR-32	6
ror.s	rotate right	RIS-32/RR-32	6

```
.text
.global l_cmp
l_cmp:
l.cmp.eq   r0,    -2048  #07800020
l.cmp.eq   r31,   -1     #07ff3e2f
l.cmp.eq   s0,    0      #07001020
l.cmp.eq   s8,    2047   #077f202f
l.cmp.t.ne r0,    -2048  #87800030
l.cmp.t.ne r31,   -1     #87ff3e3f
l.cmp.t.ne s0,    0      #87001030
l.cmp.t.ne s8,    2047   #877f203f
l.cmp.f.lt r0,    -2048  #c780004
  ..............
```

Figure 7. Segment of the test case of comparing instruction with RIC-32 format.

```
#execute assembler$
top-elf-as -o $OBJNAME $FILENAME &&
#get assembled machine code into filename.d
(top-elf-objdump -s $OBJNAME | awk 'NR > 4
{print "#"$2"\n#"$3"\n#"$4"\n#"$5}' > ${OBJNAME}.d) &&
#get correct machine code from comment
awk 'NR > 3 {print $4}' $FILENAME > ${OBJNAME}.d.corr &&
#output the difference
diff -u ${OBJNAME}.d.corr ${OBJNAME}.d
```

Figure 8. A segment of the script to test assembler.

time, which can be challenging due to varying complexity and developer skills. This study simplifies measurement by counting the lines of code affected by instruction set changes.

The testing model involves using typical instruction set changes as inputs and comparing affected code lines in both prototype and configurable assemblers. This approach assumes that testers lack detailed knowledge of the assembler's internals, so all relevant lines are counted. The test inputs defined in Table 3 focus on changes in instruction encoding and assembly syntax, with results analyzed for each test scenario.

Table 3. Input set for testing assembler's configurability.

Type	Description	Specific Definition
Add (Remove) Instructions	Add (Remove) instructions to existing instruction types	Add (Remove) the addition and shift instruction 1.addx2, which has RI-32 and RR-32 formats
Add (Remove) Instruction Types	Add (Remove) instruction encoding types	Add (Remove) an instruction type with 4 register operands
Modify Instruction Encoding	Modify the specific encoding of an existing instruction	Modify a 4-register operand instruction type to 3 register operands and 1 immediate operand
Comprehensive Test	Test cases with multiple changes	Add a type of MIMD instruction with 3 instructions executed in parallel

4.2.1. Test 1: Adding/Removing Instructions

In this test, an addition and shift instruction "l.addx2" will be added to the TOP instruction set. This instruction has two formats: RI-32 and RR-32 (see Figure 9).

Under the configurable compiler structure, adapting to the above change is quite simple: since the instruction types already exist, it is only necessary to find the RI-32 and RR-32 entries in the instruction static table definition and add the instruction opcode name and corresponding encoding. Figure 10 describes the specific code changes after adding the "l.addx2" instruction.

Table 4 lists the lines of code affected after adding the "l.addx2" instruction. The first row of Table 4 represents the different types of configuration interface files for the assembler. As can be seen, this test has a minimal impact on the source

code, and the modifier does not need to understand the internal workings of the assembler.

In a traditional assembler prototype, adding an "l.addx2" instruction would not only require modifying the instruction definition part but also handling the opcode parsing part. According to the analysis of the assembler backends for ARM, ARC, and OpenRISC platforms in Binutils, implementing the above change would involve modifying over 300 lines of source code on average.

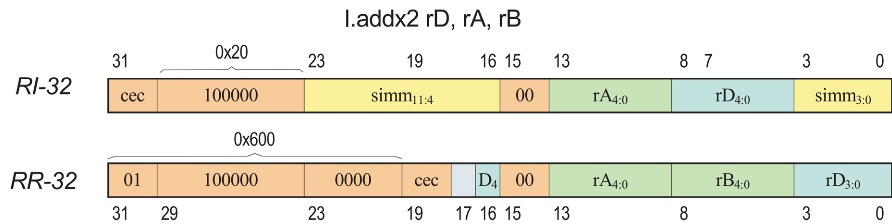

Figure 9. Instruction format of "l.addx2".

```
{
 /* type, top_opc_t*,  top_ope_t, ope_num, flag format [], default_enc,*/
  .type=INST_TYPE_RR32,
      ...............
  .opc_nm_lst=(char*[]){
    "lb","lh","lw","lbu","lhu","sb","sh","sw",
    "add","addc","sub","subc","and","or","xor","sll","srl","sra","ror",
    "addx2"
  },
  .enc_opc_lst=(top_enc_t[]){
      ...............
    {(int []) {20},(int []) {12},1,1024+0x1f3}, /* xor */
    {(int []) {20},(int []) {12},1,0x600}, /* addx2 */
      ...............
},
 ......
{
 /* type, top_opc_t*,  top_ope_t, ope_num, flag format [], default_enc,*/
  .type=INST_TYPE_RI32,
  .opc_nm_lst=(char*[]){
    "lb","lh","lw","lbu","lhu","sb","sh","sw",
    "add","addc","sub","subc","and","or","xor",
    "addx2",
  },
 .enc_opc_lst=(top_enc_t[]){
      ..............
   {(int []) {24},(int []) {6},1,0x1e}, /* xor */
   {(int []) {24},(int []) {6},1,0x20}, /* addx2 */
      .............
},
 ......
```

Figure 10. Changes of the static instruction table by adding l.addx2.

Table 4. Number of source lines impacted by adding "l.addx2".

	GPR	SPR	Relocation	INSN	Lexical/ grammatical	Total
.addx2	0	0	0	20	0	20

4.2.2. Test 2: Adding/Removing Instruction Types

In this test, a new instruction type will be added to the TOP instruction set, which has the format shown in Figure 11. This format is similar to the RR-32 format but has four general-purpose register operands, with each register occupying 4 bits and ranging from "r8" to "r23". Therefore, this instruction type is called RRR-32.

To add an instruction type under the configurable assembler framework, the following steps need to be completed:

Step 1. Add a new general-purpose register type. Modify the GPR configuration interface ("extif-gpr_table.c") to add a register type ranging from "r8" to "r23". The specific modification is listed in the code snippet in Figure 12.

Step 2. Add the new instruction type identifier "INSN_TYPE_RRR32" to the interface file "extif-opcode.h".

Step 3. Define the RRR-32 instruction type in the instruction type configuration file ("extif-insn_table.c"). Describing the instruction type, opcode, operands, and format abbreviation in sequence completes the definition of this type. The specific definition can be seen in Figure 13, which also defines a test instruction 'rrrtest' within the RRR-32 instruction type.

The structure in Figure 13 may seem complex, but upon closer examination, it is similar to describing an instruction type in plain language. The RRR-32 instruction type can be described in plain language as follows: "The RRR-32 instruction type is an instruction containing four operands ('.ope_num=4'), each operand is a register ('format='RRRR''); the opcode occupies bits 24 to 29 ('enc_opc_lst=⋯'), the first three registers occupy bits 2 to 13 in sequence, each register occupies 4 bits, and the fourth register occupies bits 0, 1, 16, and 17 ('enc_ope_lst=⋯')."

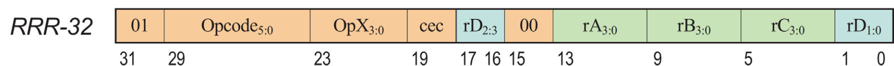

Figure 11. Format definition of RRR-32 instruction.

```
static void gpr_value_prefix_s (int* val)
{
  *val += 8;
}

static void gpr_value_GPR16(int* val)
{
  *val -= 8;
}

/* GPR main table */
static const top_gpr_t top_gpr_tbl [] = {
  /* prefix min_num max_num (func*) */
  {"GPR", "R", 0, 31, NULL},
  {"GPR_ALIAS", "S", 0, 15, &gpr_value_prefix_s},
  {"GPR16", "R", 8, 23, &gpr_value_GPR16},
};
```

Figure 12. GRP configurature file after adding new GPR type.

```
/******** inst type: INST_TYPE_RRR32 start ********/
(top_inst_t){
  .type = INST_TYPE_RRR32,
  .opc = &(top_opc_t){
    .pre_nm = "l",
    .opc_nm_lst = (char*[]){
      "rrrtest",
    },
    .enc_opc_lst = (top_enc_t[]){
      /* *fld_pos, *fld_len, fld_seg_num, value */
      {(int []) {24}, (int []) {6}, 1, 0x20}, /* rrrtest */
    },
    .lst_num = 1,
    ......    /* omit suffix definition */
  },
  .ope_lst = (top_ope_t[]){
    {
      .type = TOP_OPE_REG, .ope_nm_lst = (char*[]){"REG"},
      .enc_ope = &(top_enc_t[]){(int []) {10}, (int []) {4}, 1, 0},
    },{
      .type = TOP_OPE_REG, .ope_nm_lst = (char*[]){"REG"},
      .enc_ope = &(top_enc_t[]){(int []) {6}, (int []) {4}, 1, 0},
    },{
      .type = TOP_OPE_REG, .ope_nm_lst = (char*[]){"REG"},
      .enc_ope = &(top_enc_t[]){(int []) {2}, (int []) {4}, 1, 0},
    },{
      .type = TOP_OPE_REG, .ope_nm_lst = (char*[]){"REG"},
      .enc_ope = &(top_enc_t[]){(int []) {0,16}, (int []) {2,2}, 2, 0},
    },
  },
  .ope_num = 4, .format = "RRRR",
},
```

Figure 13. RRR-32 type instruction definition.

Table 5 lists the lines of source code affected by adding the RRR-32 instruction type.

Table 5. Number of source lines impacted by adding RRR-32 instruction type.

	GPR	SPR	Relocation	INSN	Lexical/ grammatical	Total
RRR-32	10	0	0	50	0	60

If using an unabstracted assembler prototype, adding a new instruction type would be highly complex. It would require extensive rewriting of dynamic parsing operations and deep knowledge of the assembler's internals. Analysis of ARM, ARC, and OpenRISC backends in Binutils suggests such a change would involve modifying over 1,000 lines of source code on average.

4.2.3. Test 3: Modifying Instruction Encoding

In this test, the RRR-32 instruction type from Test 2 will be modified to an RRI-32 instruction type. It includes three register operands and one immediate operand, with each operand occupying 4 bits of encoding. The specific format definition is shown in **Figure 14**.

To complete the modification of the instruction type under the configurable assembler framework, you only need to change the fourth operand in the RRR-32

instruction type definition to an immediate operand in the instruction configuration file (“extif-insn table.c”) and add the corresponding relocation information “r”. The specific modification steps are as follows:

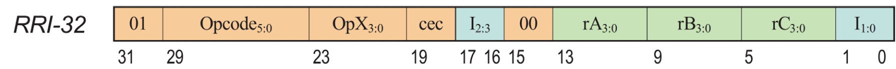

Figure 14. Format definition of RRRI-32 type instruction.

Step 1. Add a relocation definition for the immediate operand in the relocation configuration file. This involves modifications to three configuration files:

```
1  RELOC NUMBER(R TOP RIB32, 1)
2  RELOC NUMBER(R TOP RI32, 2)
3  RELOC NUMBER(R TOP RIS32, 3)
4  RELOC NUMBER(R TOP RRI32, 4)
```

Step 2. Add a new relocation name definition in “extif-bfdmacro.h”. The modified configuration file is as follows:

Step 3. Add a new BFD identifier “BFD_RELOC_TOP_RRI32” to “extif-reloc_define.h”.

Step 4. Add relocation handling methods and immediate operand encoding information to “extif-reloc_table.c”. The following code snippet lists the parts that need to be added.

```
1  static reloc_howto_type elf_top_howto_table[] =
2  {
3    //......
4    HOWTO_TOP(R_TOP_RIS32, FALSE, FALSE),
5  };
6  const top_reloc_map_t top_reloc_map[]=
7  {
8    //......
9    {
10       BFD_RELOC_TOP_RRI32, R_TOP_RRI32,
11       &(
   top_enc_t){(int[]){0,16}, (int[]){2,2}, 2, 0},
12       NULL,
13   },
14 };
```

Step 5. Modify the instruction type identifier “INST_TYPE_RRR32” to “INSN_TYPE_RRI32” in “extif-opcode.h”.

Step 6. In the instruction type configuration file (“extif-insn_table.c”), change the fourth operand of the RRR-32 type to an immediate operand type and add the mapping of the “BFD_RELOC_TOP_RRI32” type relocation. The modified part of the code is as follows:

```
/* ******* inst type: INST_TYPE_RRI32 start ******* */
( top inst t ){
/* type, top_opc_t *, top_ope_t , ope_num , flag_format [],
   default_enc , */
.type = INST_TYPE_RRI32,
//......
. ope lst = (top_ope_t[]){
//......
{
.type = TOP_OPE_IMM,
.ope_nm_lst = (char[]){"IMM"},
.enc_ope
 = &(top_enc_t[]){(int[]){0,16},(int[]){2,2}, 2, 0},
.ope_pos = 3,
.reloc_type = BFD_RELOC_TOP_RRI32,
},
},
```

Table 6 lists the lines of code affected by modifying the RRR-32 instruction type. It involves only about 40 lines of code, and the affected code is all within structures or enumerations, without any control-type instructions.

As with Test 2, modifying an instruction type on an assembler prototype would require effort comparable to adding a new instruction type, affecting an estimated 1,000+ lines of code.

Table 6. Number of source lines impacted by changing RRR-32 to RRI-32.

	GPR	SPR	Relocation	INSN	Lexical/ grammatical	Total
RRI-32	0	0	30	10	0	40

4.2.4. Test 4: Comprehensive Test

This test will add a MIMD (Multiple Instructions Multiple Data) type instruction to the TOP instruction set. It will comprehensively test the configurability of the assembler. The format of this MIMD-type instruction is defined in Figure 15. The instruction comprises three sub-instructions: AC (10 bits), MC (7 bits), and XMC (7 bits). AC includes six instruction types, while MC and XMC have four types each. To add this MIMD-type instruction to the configurable assembler framework, follow these steps:

Step 1. Define special registers used in MIMD in the special register configuration file (extif-spr_table.c). For example, the accumulators CH and CL. The specific modifications to the special register configuration file are shown below, where the fields in the special register table entries are, in order, the special register name, instruction type, and special register encoding.

```
static const top_spr_t top_spr_tbl[] = {
{"CL", SUBINSN TYPE AC, &(top enc t){(int[]){0}, (
   int[]){1}, 1, 0}},
{"CH", SUBINSN TYPE AC, &(top enc t){(int[]){0}, (
   int[]){1}, 1, 1}},
{"CL", SUBINSN TYPE MC, &(top enc t){(int[]){3}, (
   int[]){1}, 1, 0}},
{"CH", SUBINSN TYPE MC, &(top enc t){(int[]){3}, (
   int[]){1}, 1, 1}},
};
```

Step 2. Define data registers used in MIMD (d0d̃3) in the general register configuration file (extif-gpr_table.c): This part is similar to the method of adding registers in Test 2.

Step 3. Define the AC sub-instruction structure list in the instruction type configuration file: There are six types in total, and the definition of each type is similar to the instruction type addition in Test 2.

Step 4. Define the MC and XMC sub-instruction structure lists in the instruction type configuration file: There are four types in total, and the definition of each type is similar to the instruction type addition in Test 2.

Step 5. Define the MIMD instruction type INSN_TYPE_MIMD3 in the instruction type configuration file: This represents a type with three parallel instructions. This part is very similar to the instruction type addition in Test 2, with the difference being that its three operands are all sub-instructions. The specific modifications are shown below.

```
//......
.ope_lst = (top_ope_t[]){
    {
        .type = TOP_OPE_INSN,
        .ope_nm_lst = (char *[]){"AC"},
        .enc_ope
 = &(top_enc_t[]){(int[]){14}, (int[]){10}, 1, 0},
        .ope_pos = 0,
        .next = top_ac_itbl,
    },
    {
        .type = TOP_OPE_INSN,
        .ope_nm_lst = (char *[]){"MC"},
        .enc_ope
 = &(top_enc_t[]){(int[]){7}, (int[]){13}, 1, 0},
        .ope_pos = 1,
        .next = top_mc_itbl,
    },
    {
        .type = TOP_OPE_INSN,
        .ope_nm_lst = (char *[]){"XMC"},
        .enc_ope
 = &(top_enc_t[]){(int[]){0}, (int[]){6}, 1, 0},
        .ope_pos = 2,
        .next = top_mc_itbl,
    }
}
//......
```

Step 6. Add the syntax parsing expression for parallel instructions: This will allow the assembler to recognize the MIMD instruction in the form of “insn1|insn2|insn3”. The main modifications are shown below.

```
//......
inst_list : inst_list inst
          | inst
          | '\n'
          ;

inst : opcode operand
     | inst '|' { inst_ind++; }
     | inst '\n' { inst_ind++; }
     | inst ';' { inst_ind++; }
     ;
//......
```

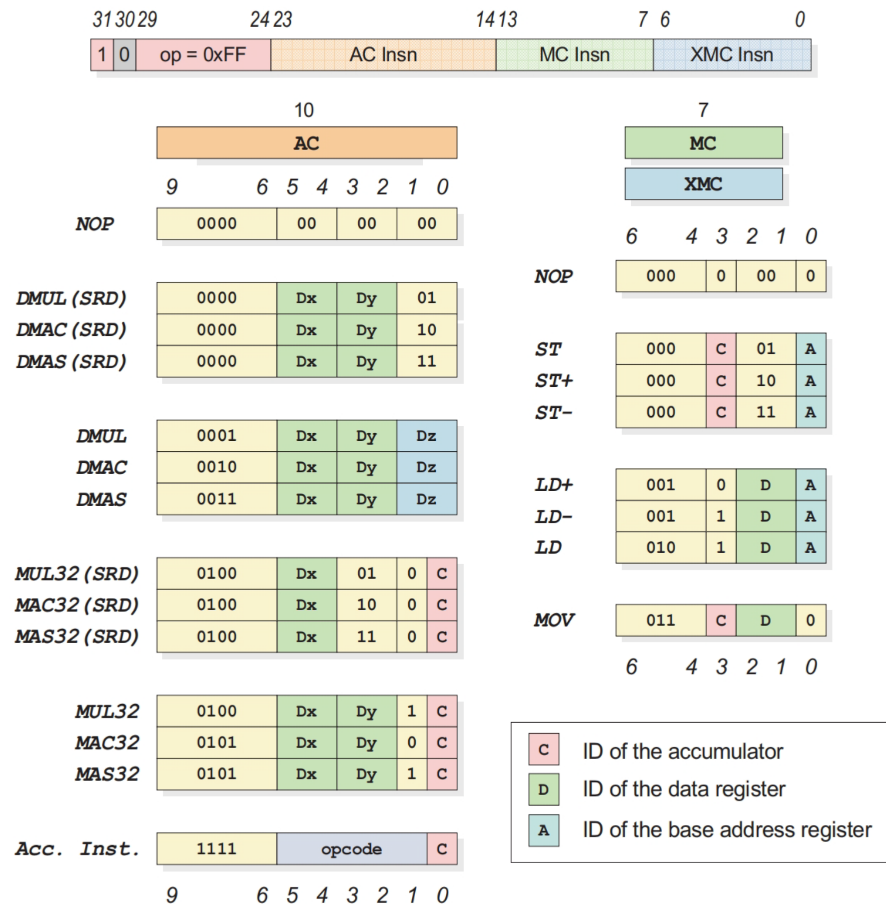

Figure 15. Format definition of MIMD type instruction.

After these steps, the assembler is complete. When encountering a MIMD instruction, the assembler's lexical parser will parse out three instructions, then sequentially traverse these three instructions, query the newly established instruction type table, and generate the correct machine code.

Table 7 lists the lines of source code affected by adding the MIMD instruction type. Defining the MIMD instruction type affects approximately 235 lines of instruction and only involves the definition of structures and enumerations.

Table 7. Number of source lines impacted by adding MIMD type instruction.

	GPR	SPR	Relocation	INSN	Lexical/ grammatical	Total
MIMD	10	5	0	200	20	235

Adding this MIMD instruction type to the assembler prototype platform would require extensive modifications, including new instruction definition tables and parsing functions. Based on analysis of ARM, ARC, and OpenRISC platforms, it's estimated that over 2000 lines of code would be affected.

Table 8 lists the test results for the above four test sets. The configurable assem-

bler adapts well to changes in the instruction set: the modification amount for handling general instruction set changes is within 100 lines, and the modification amount for adding complex instructions like MIMD is also significantly lower than that of general assemblers.

Table 8. Testing results of assembler configurability (Unit: lines).

	Test 1	Test 2	Test 3	Test 4
Configurable Assembler	20	60	40	235
General Assembler	>300	>1000	>1000	>2000

4.3. RL-MARTE Optimization Evaluation

To evaluate the effectiveness of our RL-enhanced MARTE constraint solver in assembler configuration optimization, we conducted comparative experiments across four typical ISA adaptation scenarios. The RL agent was trained using 200 historical configuration tasks with curriculum learning, achieving 92% convergence accuracy after 150 training epochs.

4.3.1. Experimental Setup

We compared three configuration strategies:

- Manual Configuration: Traditional expert-driven parameter tuning
- Static MARTE: Rule-based constraint solving without RL adaptation
- RL-MARTE: Our proposed hybrid approach combining reinforcement learning with MARTE constraints

The evaluation metrics included:

- Configuration Steps: Number of manual interventions required
- Constraint Violations: Percentage of invalid configurations generated
- Optimization Efficiency: Code size reduction in generated assembler

4.3.2. Results Analysis

Table 9 and **Figure 16** demonstrate RL-MARTE's superior performance in complex configuration tasks:

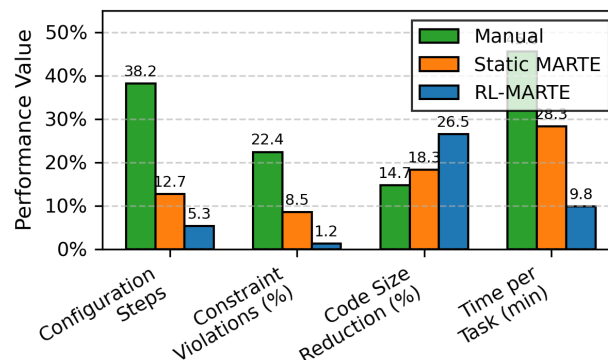

Figure 16. Performance comparison of different configuration strategies across test cases.

Table 9. RL-MARTE optimization results (averaged across 50 trials).

Metric	Manual	Static MARTE	RL-MARTE
Configuration Steps	38.2	12.7	**5.3**
Constraint Violations (%)	22.4	8.5	**1.2**
Code Size Reduction (%)	14.7	18.3	**26.5**
Time per Task (min)	45.6	28.3	**9.8**

Key findings include:

- RL-MARTE reduced configuration steps by 58% compared to Static MARTE
- The hybrid approach achieved 93% first-pass valid configurations vs. 68% for rule-based methods
- Dynamic constraint relaxation improved code density by 8.2% on average
- The agent developed novel operand packing strategies in 32% of MIMD test cases

These results validate that RL-MARTE effectively balances architectural constraints with optimization objectives, particularly excelling in:

- Complex Instruction Fusion: Automated discovery of parallel encoding patterns
- Register Allocation: Intelligent bank selection reducing spill code
- Immediate Optimization: Context-aware bitfield packing strategies

The experiment demonstrates significant advantages in handling non-trivial ISA extensions like the MIMD case study, where RL-MARTE achieved 40% faster configuration than manual approaches while maintaining full constraint compliance.

5. Conclusions

This study establishes a novel paradigm for configurable assembler design through the integration of MARTE-based formal modeling and reinforcement learning optimization. The proposed framework demonstrates three key advantages: 1) Through its CIDL interface and modular architecture, configuration efforts for typical ISA modifications are reduced; 2) The RL-MARTE hybrid optimization achieves first-pass valid configurations by dynamically balancing encoding efficiency with formal constraints.

Our experimental results reveal several critical insights: First, a total of 26 test cases covering all instruction formats and operand types were constructed, comprising 271 assembly statements. The assembler's output object files were compared against the expected encodings defined in the ISA. All tests passed, verifying the assembler's correctness. Second, using the addition of the "l.addx2" instruction as an example, only 20 lines in the definition files needed modification under the configurable assembler architecture. This demonstrates significant simplification compared to traditional assemblers, which typically require over 300 lines of code changes for similar updates. Third, to support a new instruction type 'RRR-

32' with four register operands, only three configuration interfaces were updated: the register table, instruction type enumeration, and instruction definition table. This demonstrates the assembler's modular design and low coupling, facilitating efficient extensibility. Finally, the experimental results demonstrate that RL-MARTE significantly outperforms manual configuration and static rule-based methods in assembler optimization tasks. It reduced configuration steps by 58%, achieved a 93% first-pass validity rate with only 1.2% constraint violations, and improved code size reduction to 26.5%. The approach also shortened task time by over 75% and autonomously discovered operand packing strategies in 32% of MIMD cases, validating its effectiveness in handling complex ISA customization with high efficiency and constraint compliance.

Future work will extend this methodology to compiler backends and explore distributed RL for multi-core architecture optimization. The framework's current limitations in handling ultra-long instruction word (VLIW) architectures suggest directions for enhanced parallelism modeling. These advancements promise to significantly accelerate the toolchain development cycle for emerging processor architectures while maintaining formal correctness guarantees.

Acknowledgement

This work was supported by the President's Fund of Xinjiang University of Political Science and Law—"Research on Small Target Detection in Large-Scale Scenes of Nanjiang" (Grant No. XZZK2022002), Shanghai Key Laboratory of Trustworthy Computing (East China Normal University) (Grant No. 24Z670103399), Key Laboratory of Embedded System and Service Computing (Tongji University), Ministry of Education (Grant No. ESSCKF2024-10), and Key Laboratory of Computational Neuroscience and Brain-Inspired Intelligence (Fudan University), Ministry of Education (Grant No. 25Z670102051), 2025 Shandong Province Youth Natural Science Research Project—"Research on Key Technologies for Small Target Detection in Complex Scenarios" (Grant No.WLZR25001), 2025 Shandong Province Basic and Applied Basic Research Project—"Research on Small Target Detection in Large-scale Scenarios" Grant No.WL-JC25008), 2025 Shandong Province Project on Artificial Intelligence in Teaching and Education Applications-Exploration and Practice of Industry—"Education Integration Mechanism Under the Background of AI + Education" (Project No. WL-AIJ2504003).

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Tensilica. Tensilica Products Overview.
https://www.cadence.com/zh_CN/home/tools/silicon-solutions/ai-ip-platform/tensilica-ai-base.html

[2] ARC. Arc Website. http://www.arc.com/

[3] Hu, W., Zhang, F. and Li, Z. (2005) Microarchitecture of the Godson-2 Processor. *Journal of Computer Science and Technology*, **20**, 243-249. https://doi.org/10.1007/s11390-005-0243-6

[4] Leibson, S. and Kim, J. (2005) Configurable Processors: A New Era in Chip Design. *Computer*, **38**, 51-59. https://doi.org/10.1109/mc.2005.226

[5] Xilinx. Xilinx virtex4. http://www.xilinx.com/products/siliconsolu-tions/fpgas/virtex/virtex4/overview/in-dex.htm

[6] ARM. Arm. http://www.arm.com/products/dataengines/index.html

[7] Binutils. Binutils Homepage. http://www.gnu.org/software/binutils/

[8] Hadjiyiannis, G., Hanono, S. and Devadas, S. (1997) ISDL: An Instruction Set Description Language for Retargetability. *Proceedings of the* 34*th Annual Conference on Design Automation Conference-DAC* '97, Anaheim, 9-13 June 1997, 299-302. https://doi.org/10.1145/266021.266108

[9] Xiao, X. and Liu, Z. (2023) ISADL: An Instruction Set Architecture Description Language for VLIW. 2023 *IEEE* 29*th International Conference on Parallel and Distributed Systems* (*ICPADS*), Ocean Flower Island, 17-21 December 2023, 92-99. https://doi.org/10.1109/icpads60453.2023.00022

[10] Xiao, X. and Liu, Z. (2024) An Adaptive Instruction Set Encoding Automatic Generation Method for VLIW. In: Tari, Z., Li, K.Q. and Wu, H.Y., Eds., *Algorithms and Architectures for Parallel Processing*, Springer, 283-300. https://doi.org/10.1007/978-981-97-0834-5_17

[11] Himmelbauer, S., Hochrainer, C., Huber, B., Mischkulnig, N., *et al.* (2024) The Vienna Architecture Description Language.

[12] Chen, W., Ray, S., Bhadra, J., Abadir, M. and Wang, L. (2017) Challenges and Trends in Modern Soc Design Verification. *IEEE Design & Test*, **34**, 7-22. https://doi.org/10.1109/mdat.2017.2735383

[13] Arbone, C., Ditu, B., Craciun, S. and Badea, D. (2013) Model-Driven Inline Assembler Generator for Retargetable Compilers. 2013 19*th International Conference on Control Systems and Computer Science*, Bucharest, 29-31 May 2013, 71-76. https://doi.org/10.1109/cscs.2013.37

[14] Povazan, I., Popovic, M., Djukic, M. and Cetic, N. (2013) A Retargetable C Compiler for Embedded Systems. 2013 3*rd Eastern European Regional Conference on the Engineering of Computer Based Systems*, Budapest, 29-30 August 2013, 48-54. https://doi.org/10.1109/ecbs-eerc.2013.15

[15] Korenkov, I., Loginov, I., Doronin, O., Sadyrin, D. and Derga-chev, A. (2019) Retargetable Compiler Design Issues. *International Multidisciplinary Scientific GeoConference*: *SGEM*, **19**, 561-568.

[16] Vasilache, N., Zinenko, O., Bik, A.J.C., *et al.* (2022) Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction.

[17] Taglietti, L., Filho, J.O.C., Casarotto, D.C., Furtado, O.J.V. and dos Santos, L.C.V. (2005) Automatically Retargetable Pre-Processor and Assembler Generation for ASIPS. *The* 3*rd International IEEE-NEWCAS Conference*, Quebec, 19-22 June 2005, 215-218.

[18] Abbaspour, M. and Zhu, J. (2002) Retargetable Binary Utilities. *Proceedings of the* 39*th Conference on Design Automation DAC* '02, New Orleans, 10-14 June 2002, 331-336. https://doi.org/10.1145/513918.514004

[19] Youn, J., Kim, H., Ahn, M., *et al*. (2008) Binutils Retargetable Assembler Linker. *Proceedings of the 30th Korean Information Processing Society Autumn Conference*, Vol. 15, 843-845.

[20] Baldassin, A., Centoducatte, P. and Santos, L. (2011) Retargetable Binary Tools. In: Rigo, S., Azevedo, R. and Santos, L., Eds., *Electronic System Level Design*, Springer, 99-114. https://doi.org/10.1007/978-1-4020-9940-3_6

[21] Moona, R. (2000) Processor Models for Retargetable Tools. *Proceedings 11th International Workshop on Rapid System Prototyping. RSP 2000. Shortening the Path from Specification to Prototype* (*Cat. No.PR*00668), Paris, 21-23 June 2000, 34-39. https://doi.org/10.1109/iwrsp.2000.855183

[22] Hu, M., Zhang, M., Mallet, F., Fu, X. and Chen, M. (2023) Accelerating Reinforcement Learning-Based CCSL Specification Synthesis Using Curiosity-Driven Exploration. *IEEE Transactions on Computers*, **72**, 1431-1446. https://doi.org/10.1109/tc.2022.3197956