

# LLM-based Generation of High-Level Benchmarks for MVL Designs

Rolf Drechsler

University of Bremen/DFKI, 28359 Bremen, Germany  
drechsler@uni-bremen.de

**Abstract**—Benchmark circuits are crucial for evaluating and comparing algorithms in *Electronic Design Automation* (EDA). While extensive benchmark suites exist for binary logic, only a few are available for *Multi-Valued Logic* (MVL) circuits. Existing methods often rely on encoded binary circuits, which do not adequately capture the structural properties of MVL designs.

In this work, we demonstrate that *Large Language Models* (LLMs) can be used to generate meaningful MVL benchmarks with minimal effort on a higher abstraction level. As a proof of concept, we present the automatic generation of an 8-bit *Arithmetic Logic Unit* (ALU) operating over GF(3). The resulting benchmark exhibits realistic design structures and functional diversity, illustrating the potential of LLM-based generation for future MVL synthesis and verification research.

**Index Terms**—multi-valued logic, MVL, benchmarks, large language models, LLMs, synthesis, verification

## I. INTRODUCTION

Benchmarking plays a central role in the evaluation and comparison of newly proposed algorithms in EDA. Only through well-defined and widely accepted benchmark suites can the performance, robustness, and scalability of methods be objectively assessed. While synthetic test cases or random functions may serve as initial indicators, they are generally insufficient for a meaningful evaluation. Instead, benchmarks should reflect structures that actually occur in digital designs, such as adders, multipliers, or complete ALUs.

Since the early 1980s, a variety of benchmark sets have been proposed to address this need. One of the earliest and most influential examples are the C-benchmarks [1], which have subsequently been refined and extended in multiple directions (see e.g. [2]). Although originally developed for testing algorithms, these benchmarks quickly found application in other areas of EDA, including logic synthesis, optimization, and formal verification. Similar developments can be observed in adjacent domains: in reversible computing, for instance, the availability of dedicated benchmark circuits [3] and layout structures [4] has likewise driven progress by providing common reference points for evaluation.

In contrast, for MVL circuits, only a limited number of benchmark sets exist. Often, binary benchmarks are reused by grouping input variables and applying suitable encodings to emulate multi-valued behavior. However, such transformations fail to capture the inherent characteristics of MVL designs and thus do not reflect their specific structural and functional properties [5]. An alternative approach, based on decision diagrams, was proposed in [6], enabling the generation of

MVL circuits at a low structural level. While effective for specific use cases, this method remains limited in terms of scalability and abstraction.

Recently, LLMs have become available as powerful tools for knowledge-based generation tasks and have found applications in multiple areas of EDA, including design, testing, and verification (for an overview, see e.g. [7]). LLM-based approaches leveraging hierarchical reasoning [8] or abstract, software-level descriptions [9] have demonstrated their potential for automatic test case generation and design synthesis.

In this paper, we show that, based on LLMs, meaningful benchmark circuits for multi-valued logic can be generated with remarkably low effort. Our approach exploits the semantic and structural capabilities of modern LLMs to produce benchmarks that not only resemble realistic design patterns but also maintain the distinctive properties of MVL systems. As an example, an 8-bit ALU is discussed, where it is shown that with very few prompts a complete simulation environment can be generated and executed. This opens a new pathway for the systematic evaluation and comparison of algorithms in MVL synthesis, optimization, and verification.

The paper is structured as follows: In Section II it is shown how based on simple prompts an ALU for GF(3) can be generated. The resulting output is shown in Section III, where the complete code in compilable C is provided including an experimental evaluation. Further details how this approach can be applied and extended are discussed in Section IV. Finally, in Section V the results are summarized and future work is outlined.

## II. PROMPT-BASED BENCHMARK GENERATION

Following the idea of [10], where a simple MVL ALU was designed in SystemC and then simulated, in the following we generate an 8-bit ALU in MVL. For compactness, the prompts are shown that were given to ChatGPT-5<sup>1</sup>, while the answers are summarized for shortness of the presentation and briefly discussed.

**Prompt:** Can you give me a textual specification of an 8-bit ALU?

A complete ALU is generated in VHDL, with I/O ports as shown in Table I and II, respectively.

Unsurprisingly, it is in 2-valued logic. Therefore, the next step is to generalize to MVL.

<sup>1</sup><https://openai.com/>

TABLE I  
INPUTS OF THE 8-BIT ALU

Signal	Description
A[7:0]	First 8-bit operand
B[7:0]	Second 8-bit operand
OP[3:0]	4-bit operation code (selects the function to perform)
Cin	Carry-in (used for addition/subtraction with carry)

TABLE II  
OUTPUTS OF THE 8-BIT ALU

Signal	Description
R[7:0]	8-bit result of the operation
Cout	Carry-out (bit beyond the MSB in addition, or borrow in subtraction)
Zero (Z)	Set if result = 0
Negative (N)	Set if MSB of result = 1 (two's complement sign bit)
Overflow (V)	Set if signed overflow occurs in arithmetic operations
Carry (C)	Same as Cout, but stored for status register use

**Prompt:** Can you design an 8-bit ALU, but not in 2-valued, but in 3-valued logic?

Since the logic is not clearly specified, the system provides this as a feedback and mentions alternatives, like Kleene logic (K3) with the logic elements 0, 1 and X. While this is a 3-valued modelling, it still addresses binary circuits. Also proposed are Lukasiewicz logic (L3) and Kleene strong logic: widely used in digital design to model 0, 1 and U (unknown/high-impedance). Also the most likely application is mentioned by the system: *For hardware verification, usually it's {0, 1, U}*.

**Prompt I** want a calculation over GF(3).

Presented options for the ALU are integer style (treating operands as numbers 0...6560 and doing addition/multiplication modulo  $3^8$ ), or in the vector style (8 parallel trit operations, like eight independent 1-trit ALUs).

**Prompt** Please in integer style.

To finalize the design and check the output, a simulation environment is needed.

**Prompt** Can you simulate the design for 20 random test vectors?

And to also allow for an executable specification, here C code is generated.

**Prompt** Please show code in C.

The generated C code (see next section) can be fully compiled without any further modifications.

### III. CASE STUDY: 8-BIT ALU OVER GF(3) IN C

In this section the generated C code is presented and discussed.<sup>2</sup> For the experimental evaluation Pelles C<sup>3</sup> was used.

<sup>2</sup>The code is not modified, only the formatting is slightly changed to better fit in the two-column format of the paper.

<sup>3</sup><http://www.smorgasbordet.com/pelles/>

```
// alu_gf3.c
// C model of an 8-trit ALU over GF(3)
// in integer style (mod 6561)
// + random testbench.
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>
#include <stdbool.h>
```

```
enum {
    OP_ADD = 0x0,
    OP_SUB = 0x1,
    OP_MUL = 0x2,
    OP_NEG = 0x3,
    OP_INC = 0x4,
    OP_DEC = 0x5,
    OP_PASSA = 0x6,
    OP_PASSB = 0x7
};
```

```
#define MOD 6561u // 3^8
#define HALF ((MOD - 1u) / 2u) // 3280
```

```
typedef struct {
    bool Z; // zero
    bool N; // "negative"
            // (upper half of range)
    bool C; // carry for ADD
            // (wrap indicator)
    bool V; // overflow
            // (unused here; 0)
} Flags;
```

```
static inline uint32_t
mod6561(int64_t x) {
    // Normalize to [0, MOD-1]
    int64_t r = x % MOD;
    if (r < 0) r += MOD;
    return (uint32_t)r;
}
```

Fig. 1. Initialization

In Figure 1 the initial setup is shown. This includes the operators that are provided as an enum. Please notice that the dialog in the previous section did not specify the operators. The system proposed a simple ALU with addition, subtraction, etc. Of course, further operators can be included - either by more specific prompts or by modifications of the generated code. Information on the basic field GF(3), representing trits, as well as on internal flags is provided.

The proposed ALU performs all operations over the finite ring  $Z_{3^8}$ , enabling arithmetic on 8-trit operands represented as integers in the range  $0 \leq x < 3^8 = 6561$ . Unlike conventional

binary ALUs that operate modulo  $2^n$ , this design employs base-3 arithmetic, thus supporting ternary computation directly at the architectural level. Each arithmetic operation is computed using modular arithmetic with a single modulus reduction, ensuring that results always remain within the valid range without overflow or undefined states.

The ALU (see Figures 2 and 3) supports a minimal yet expressive instruction set, including addition, subtraction, multiplication, negation, increment, decrement, and operand pass-through. The operations are defined as

$$R = (A \pm B) \bmod 3^8, \quad R = (A \times B) \bmod 3^8,$$

and analogously for unary functions. This design choice allows direct implementation using integer arithmetic on conventional digital platforms while maintaining mathematical equivalence to computations over  $Z_{3^8}$ .

Four status flags accompany each operation: the *zero flag* ( $Z$ ), set when the result equals zero; the *negative flag* ( $N$ ), set when the result exceeds half the modulus ( $R > 3280$ ), reflecting a pseudo-signed interpretation of balanced ternary representation; the *carry flag* ( $C$ ), asserted during additive wrap-around beyond the modulus; and the *overflow flag* ( $V$ ), reserved for potential extensions that support full balanced-ternary arithmetic. These flags facilitate higher-level control flow and conditional execution in ternary architectures.<sup>4</sup>

The design’s primary advantages are its *algebraic closure*, *deterministic wrap-around behavior*, and *implementation simplicity*. Because all operations are performed modulo  $3^8$ , no exceptional or undefined states occur, and arithmetic can be executed using compact, purely combinational logic. This makes the ALU particularly well suited for research into non-binary computing systems, algebraic hardware modeling, and energy-efficient arithmetic in emerging technologies such as multi-valued logic circuits and quantum-inspired architectures.

However, the approach also introduces certain limitations. The ALU does not directly implement Boolean operations such as AND, OR, or XOR, which must instead be reformulated in algebraic terms within  $GF(3)$ . Moreover, practical hardware realization requires ternary or multi-level logic cells, which are more complex than standard CMOS binary gates. Despite these constraints, the presented ALU provides a mathematically rigorous and compact framework for evaluating base-3 arithmetic in computational and hardware design studies.

The `main()`-program (see Figure 4) implements a simple simulation environment executing 20 random evaluations of the ALU. The output of a run is shown in Figure 5.

## IV. DISCUSSION

While the previous sections have shown how a simple ALU benchmark can be generated at very low cost, in the following it is discussed what additional steps have to be considered for complete benchmark suites.

<sup>4</sup>Since the initial ALU was two-valued and was then “converted” to  $GF(3)$ , not all flags are supported (see last lines of Figure 2). Also, not all operands set the corresponding flags, and corrections are needed for a complete design.

### A. Scalability of Datapath

One of the major advantages of the proposed approach lies in its inherent scalability. Because the benchmark circuits are generated through structured descriptions, it is straightforward to extend the datapath width from smaller designs, such as 8-bit ALUs, to larger configurations with 16 or even 32 bits. This scalability is not only important from a practical perspective - where ALUs of different sizes appear in numerous application contexts - but it also enables the systematic study of how EDA tools behave when circuit size increases. For instance, the synthesis and verification runtimes, as well as the resource requirements, can be analyzed as functions of datapath width, providing valuable insights into algorithmic efficiency and scalability. Furthermore, the proposed method allows for scaling in much finer steps, such as from 8 to 9, 10, or 11 bits, which is typically difficult to achieve using conventional, manually created benchmarks. Such gradual scaling is particularly helpful for the controlled evaluation of synthesis and optimization techniques, where smooth transitions between benchmark sizes facilitate detailed analysis of tool performance and scalability limits.

### B. Scalability of Multi-Valued Logic

In addition to datapath scalability, the proposed approach naturally extends to designs operating over different numbers of logic values ( $k$ ). This means that multi-valued logic circuits can be generated for various domains, such as ternary ( $k = 3$ ), quaternary ( $k = 4$ ), or higher-valued systems. Having benchmark circuits for several values of  $k$  is essential to systematically investigate how algorithmic performance changes with increasing logical complexity. For example, decision diagrams, synthesis heuristics, or verification algorithms might scale differently when moving from binary to higher-valued representations. The flexibility to generate families of functionally equivalent but differently valued circuits provides a unique opportunity to study the relationship between the theoretical properties of the logic and the practical limitations of current EDA tools. Such insights could be used to guide the development of new algorithms that are explicitly optimized for multi-valued or finite-field logic systems.

### C. Choice of Logic

The generation approach is not restricted to a specific form of multi-valued logic. Depending on the intended application, alternative logical systems can be employed, including Łukasiewicz logic ( $L_3$ ), Kleene’s strong logic, or logics defined over finite fields such as Galois Fields (GFs). This diversity enables the creation of benchmark sets that reflect a wide spectrum of reasoning and algebraic paradigms. Moreover, the generated circuits allow researchers to systematically assess how well existing decision procedures and algorithmic frameworks perform for each of these logics. For example, the benchmark proposed in this work can be used to evaluate algorithms previously introduced for GF-based multi-valued designs, such as those discussed in [11], [12]. In this way, the availability of standardized benchmarks across different logical domains can foster a deeper understanding of the

```

uint32_t alu_exec(uint32_t A, uint32_t B, uint32_t op, Flags *f)
{
    // Constrain inputs to 0..6560
    A %= MOD;
    B %= MOD;

    uint32_t R = 0;
    bool cflag = false;

    switch (op & 0xF) {
        case OP_ADD: {
            uint32_t sum = A + B;          // max 13120 < 2^16
            R = sum % MOD;
            cflag = (sum >= MOD);         // wrap occurred
        } break;
        case OP_SUB:
            R = mod6561((int64_t)A - (int64_t)B);
            break;
        case OP_MUL: {
            uint64_t prod = (uint64_t)A * (uint64_t)B; // max ~43e6 < 2^32
            R = (uint32_t)(prod % MOD);
        } break;
        case OP_NEG:
            R = mod6561(-(int64_t)A);
            break;
        case OP_INC:
            R = (A + 1u) % MOD;
            break;
        case OP_DEC:
            R = mod6561((int64_t)A - 1);
            break;
        case OP_PASSA:
            R = A;
            break;
        case OP_PASSB:
            R = B;
            break;
        default:
            R = 0;
            break;
    }

    if (f) {
        f->Z = (R == 0u);
        f->N = (R > HALF); // treat upper half as "negative"
                          // (balanced ternary mode would refine this)
        f->C = cflag;      // only meaningful for ADD here
        f->V = false;     // not defined for pure modular arithmetic
    }
    return R;
}

```

Fig. 2. Structure of ALU

```

static const char* opname(uint32_t op) {
    switch (op & 0xF) {
        case OP_ADD:    return "ADD";
        case OP_SUB:    return "SUB";
        case OP_MUL:    return "MUL";
        case OP_NEG:    return "NEG";
        case OP_INC:    return "INC";
        case OP_DEC:    return "DEC";
        case OP_PASSA:  return "PASSA";
        case OP_PASSB:  return "PASSB";
        default:        return "NOP";
    }
}

```

Fig. 3. Available operators

strengths and weaknesses of current EDA methodologies and stimulate the development of more general frameworks that handle heterogeneous logic representations.

#### D. Translation to Other Description Languages

While the current example has been presented in the C programming language, the methodology is not tied to this specific representation. Because the generation process is based on high-level semantic descriptions, it can easily be extended to other system or hardware description languages such as Python, SystemC, Verilog, or VHDL. This translation capability broadens the applicability of the approach and allows the generated benchmarks to be used across different toolchains and design environments. For instance, C and Python representations may be used to evaluate High-Level Synthesis (HLS) frameworks, while Verilog or VHDL versions can serve as input for logic synthesis or gate-level simulation tools. Furthermore, these benchmarks can play an important role in the domain of formal verification, where equivalence checking or model checking requires meaningful and structurally rich test cases. Thus, the proposed approach provides a bridge between software-level modeling and hardware-level implementation, enabling consistent evaluation across the design hierarchy.

#### E. Translation to Other Abstraction Levels

Traditional benchmark collections typically focus on a single level of abstraction, such as gate-level or behavioral descriptions. However, in modern EDA research, it is increasingly important to analyze the consistency of tools and algorithms across multiple abstraction layers. The approach presented in this paper allows the generation of equivalent descriptions of the same design at different abstraction levels. For example, a high-level behavioral description in C or SystemC can be complemented by a functionally equivalent VHDL or Verilog implementation, which in turn can be synthesized into a structural netlist composed of basic logic gates. This hierarchical structure enables detailed studies of refinement, synthesis accuracy, and verification consistency between abstraction levels. Such benchmarks are valuable

not only for traditional synthesis evaluation but also for formal equivalence checking, high-level synthesis validation, and model-based verification tasks.

## V. CONCLUSION

Overall, the proposed LLM-based approach for generating multi-valued logic benchmarks offers a high degree of flexibility, scalability, and adaptability. It supports variations in datapath width, number of logic values, logical systems, and description languages, as well as the generation of functionally consistent representations at multiple abstraction levels. These properties make it a powerful foundation for comprehensive and reproducible evaluation of EDA algorithms across diverse domains. While it has been shown that the generated benchmark can be compiled and is functionally, further extensions, like e.g. additional commands, can be added either by modifying the prompts in the LLM or by changing the generated source code.

In the future, expanding this approach to include automatic parameterization, structural diversity, and integration with open-source EDA frameworks could further enhance its usefulness. Ultimately, the combination of LLM-driven generation and systematic benchmark design has the potential to accelerate research on multi-valued synthesis, optimization, and verification by providing standardized, easily extensible, and semantically rich benchmark sets.

## REFERENCES

- [1] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational circuits and a target translator in fortran," in *Int'l Symp. Circ. and Systems, Special Sess. on ATPG and Fault Simulation*, 1985, pp. 663–698.
- [2] F. Brglez and R. Drechsler, "Design of experiments in CAD: Context and new data sets for ISCAS'99," in *Int'l Symp. Circ. and Systems*, vol. VI, 1999, pp. 424–427.
- [3] R. Wille, D. Grosse, L. Teuber, G. Dueck, and R. Drechsler, "RevLib: An online resource for reversible functions and reversible circuits," in *38th International Symposium on Multiple Valued Logic*, 2008, pp. 220–225.
- [4] P. Verplaetse, J. V. Campenhout, and D. Stroobandt, "On synthetic benchmark generation methods," in *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4, 2000, pp. 213–216 vol.4.
- [5] R. Brayton and S. Khatri, "Multi-valued logic synthesis," in *Proceedings Twelfth International Conference on VLSI Design*, 1999, pp. 196–205.
- [6] M. Radmanovic and R. Stankovic, "Generating synthetic MVL benchmarks from random MDDs under restrictions," in *2018 IEEE 48th International Symposium on Multiple-Valued Logic (ISMVL)*, 2018, pp. 168–173.
- [7] C. K. Jha, M. Hassan, K. Qayyum, S. Ahmadi-Pour, K. Xu, R. Qiu, J. Blocklove, L. Collini, A. Nakkab, U. Schlichtmann, G. Zhang, R. Karri, B. Li, S. Garg, and R. Drechsler, "Large language models (LLMs) for verification, testing, and design," in *IEEE European Test Symposium*, 2025, pp. 1–10.
- [8] S. G. Lopes, S. Zhu, and G. Alonso, "Exploring large language models for hierarchical hardware circuit and testbench generation," in *ACM Transactions on Design Automation of Electronic Systems*, 2025.
- [9] E. Farchi, S. Froimovich, R. Katan, and O. Raz, "Automatic generation of benchmarks and reliable LLM judgment for code tasks," in *arXiv:2410.21071*, 2024.
- [10] D. Große, G. Fey, and R. Drechsler, "Modeling multi-valued circuits in SystemC," in *Int'l Symp. on Multi-Valued Logic*, 2003, pp. 281–286.
- [11] R. Stankovic and R. Drechsler, "Circuit design from kronecker galois field decision diagrams for multiple-valued functions," in *Proceedings of the 27th International Symposium on Multiple-Valued Logic*, 1997.
- [12] S. Morioka, Y. Katayama, and T. Yamane, "Towards efficient verification of arithmetic algorithms over galois fields  $gf(2^m)$ ," in *Proceedings of the 13th International Conference on Computer Aided Verification*, 2001.

```

int main(void)
{
    srand((unsigned)time(NULL));

    for (int i = 0; i < 20; ++i) {
        uint32_t A = (uint32_t)(rand() % MOD);
        uint32_t B = (uint32_t)(rand() % MOD);
        uint32_t op = (uint32_t)(rand() % 8); // choose among 0..7

        Flags f = {0};
        uint32_t R = alu_exec(A, B, op, &f);

        printf("Test %3d: OP=%-6s A=%4u B=%4u -> R=%4u  " "Z=%d N=%d C=%d V=%d\n",
            i, opname(op), A, B, R, f.Z, f.N, f.C, f.V);
    }

    return 0;
}

```

Fig. 4. Main program

```

Test  0: OP=PASSB  A= 329 B=3787 -> R=3787  Z=0 N=1 C=0 V=0
Test  1: OP=SUB    A=4320 B=6190 -> R=4691  Z=0 N=1 C=0 V=0
Test  2: OP=NEG    A=2332 B=3913 -> R=4229  Z=0 N=1 C=0 V=0
Test  3: OP=DEC    A=3765 B=4845 -> R=3764  Z=0 N=1 C=0 V=0
Test  4: OP=PASSB  A=6121 B= 537 -> R= 537  Z=0 N=0 C=0 V=0
Test  5: OP=ADD    A=1687 B=5696 -> R= 822  Z=0 N=0 C=1 V=0
Test  6: OP=ADD    A=6184 B=3787 -> R=3410  Z=0 N=1 C=1 V=0
Test  7: OP=PASSA  A=4132 B=1606 -> R=4132  Z=0 N=1 C=0 V=0
Test  8: OP=PASSA  A=1415 B=2947 -> R=1415  Z=0 N=0 C=0 V=0
Test  9: OP=INC    A=2207 B=4983 -> R=2208  Z=0 N=0 C=0 V=0
Test 10: OP=INC    A=4410 B=2980 -> R=4411  Z=0 N=1 C=0 V=0
Test 11: OP=MUL    A=5060 B= 456 -> R=4449  Z=0 N=1 C=0 V=0
Test 12: OP=MUL    A=3832 B=6114 -> R=6078  Z=0 N=1 C=0 V=0
Test 13: OP=ADD    A=3630 B=6236 -> R=3305  Z=0 N=1 C=1 V=0
Test 14: OP=NEG    A=5698 B=3538 -> R= 863  Z=0 N=0 C=0 V=0
Test 15: OP=MUL    A= 933 B= 849 -> R=4797  Z=0 N=1 C=0 V=0
Test 16: OP=PASSA  A=4883 B=1747 -> R=4883  Z=0 N=1 C=0 V=0
Test 17: OP=PASSB  A=1274 B=2406 -> R=2406  Z=0 N=0 C=0 V=0
Test 18: OP=MUL    A=2953 B=5314 -> R=4891  Z=0 N=1 C=0 V=0
Test 19: OP=DEC    A=5448 B=1194 -> R=5447  Z=0 N=1 C=0 V=0

```

Fig. 5. Output of simulation