Cocotb Log Parser Documentation

Overview

The Cocotb Log Parser is a Python tool designed to analyze testbench logs from RISC-V processor verification. It extracts test case information, tracks instruction patterns, register usage, and provides comprehensive statistics for verification analysis.

Features

Data Extraction

- **Test Case Information**: Test numbers, instruction types, names, and execution results
- **Register Analysis**: Usage patterns for destination (rd) and source registers (rs1, rs2)
- **Immediate Values**: Enhanced tracking of immediate operands with multiple format support
- **Value Range Analysis**: Categorization of values into power-of-2 ranges
- **Status Tracking**: Success/failure rates, overflow conditions (properly handled), illegal instructions

Statistical Analysis

- Instruction frequency counting
- Register usage distribution
- Immediate value distribution and range analysis
- Value range distribution analysis
- Success rate calculation with proper overflow handling
- Failure pattern identification

Export Capabilities

- JSON export for further analysis
- Detailed failure reports
- Instruction-specific analysis including immediate value patterns

Installation

```bash
# No external dependencies required - uses only Python standard library
python3 parser.py --help
```

Usage

Basic Usage

```bash
# Parse a log file and show summary
python3 parser.py testbench.log

# Show detailed failure analysis
python3 parser.py testbench.log --failures

# Analyze a specific instruction
python3 parser.py testbench.log --analyze add

# Export results to JSON
python3 parser.py testbench.log --export results.json
```

Command Line Arguments

- `logfile`: Path to the cocotb log file to parse (required)
- `--failures`: Show detailed information about failed test cases
- `--analyze INSTRUCTION`: Perform detailed analysis of a specific instruction
- `--export FILENAME`: Export all results to a JSON file

Expected Log Format

The parser expects log entries in the following format:

```
Test 2239
Instruction type: R-Type
Name: add
Registers: rd=20, rs1=8 rs2=18
Instruction: 00000001001001000000101000110011
Pre-instruction: rd=545, rs1=-1, rs2=-2048
Actual: rd=-2049, rs1=-1, rs2=-2048
Expected rd: -2049
Success!
```

Required Fields

- **Test Number**: `Test XXXX`

- **Instruction Type**: `Instruction type: R-Type|I-Type|...`
- **Instruction Name**: `Name: add|sub|addi|...`
- **Status**: `Success!` or failure indicators

Optional Fields

- **Registers**: `Registers: rd=X, rs1=Y rs2=Z` or `rd=X, rs1=Y imm=Z`
- **Binary Instruction**: `Instruction: XXXXXXXX...`
- **Pre-execution Values**: `Pre-instruction: rd=X, rs1=Y, rs2=Z`
- **Actual Results**: `Actual: rd=X, rs1=Y, rs2=Z`
- **Expected Results**: `Expected rd: X` or `Expected rd: overflow`
- **Status Flags**: `overflow`, `illegal`, etc.

Enhanced Immediate Value Support

The parser now supports multiple immediate value formats:

- `imm=123` (in Registers line)
- `immediate: 123`
- `imm: 123`
- `immediate = 123`
- `imm = 123`
- `Immediate: 123`
- `IMM: 123`

All formats are case-insensitive and support negative values.

Overflow Handling (Updated)

Important Change: Overflow conditions are now properly handled and do not automatically count as failures. The parser uses improved logic to determine test success:

Success Determination Logic

A test is considered successful if ANY of these conditions are met:

1. **Explicit Success**: The log contains `Success!`
2. **Expected Overflow**: The test expects overflow (`Expected rd: overflow`) AND overflow actually occurred

3. **Value Match**: For non-overflow cases, the actual result matches the expected result

Overflow Status

- Overflows are tracked separately from failures
- A test can have overflow AND still be marked as successful
- Overflow counts are displayed as "counted as successes when expected"

Output Analysis

Summary Report

The parser generates a comprehensive summary including:

Test Results

- Total number of tests executed
- Success/failure counts and percentages
- Overflow occurrences (with note that overflows can be successful)
- Illegal instruction counts

Instruction Statistics

- Most frequently executed instructions
- Instruction type distribution (R-Type, I-Type, etc.)
- Per-instruction success rates

Register Usage Analysis

- Overall register usage frequency
- Destination register (rd) usage patterns
- Source register usage patterns (rs1, rs2)

Value Analysis (Enhanced)

- **Immediate Value Statistics**: Complete tracking and analysis
    - Total instructions with immediate values
    - Most common immediate values
    - Immediate value range distribution
- Register value range distribution

- Pre-instruction and result value patterns

Value Range Categorization

Values are automatically categorized into ranges for pattern analysis:

Range	Description	Typical Use Case
zero	Value = 0	Zero register, cleared values
±2^4 (pos/neg)	1 to 16	Small constants, loop counters
±2^8 (pos/neg)	17 to 256	Byte values, small offsets
±2^10 (pos/neg)	257 to 1024	Small immediate fields
±2^12 (pos/neg)	1025 to 4096	RISC-V I-type immediates
±2^16 (pos/neg)	4097 to 65536	16-bit values
±2^20 (pos/neg)	65537 to 1M	RISC-V U-type immediates
±2^31 (pos/neg)	1M+ to 2G	Large 32-bit values
large (pos/neg)	> 2^31	Very large values

Failure Analysis

When using `--failures`, the parser provides detailed information about each failed test:

```
=== Test 1245 FAILED ===
  Instruction: add (R-Type)
  Registers: rd=20, rs1=8, rs2=18
  Immediate: 42
  Expected rd: -2049, Actual rd: 2047
  Status: OVERFLOW
  Raw: Instruction type: R-Type...
```

Key Points:

- Only tests that are genuinely incorrect are shown as failures
- Overflow conditions that match expectations are NOT shown as failures
- Immediate values are now properly displayed when present

API Reference

TestCase Class

```
python
```

```
@dataclass
class TestCase:
    test_number: int
    instruction_type: Optional[str] = None
    instruction_name: Optional[str] = None
    rd: Optional[int] = None          # Destination register number
    rs1: Optional[int] = None         # Source register 1 number
    rs2: Optional[int] = None         # Source register 2 number
    immediate: Optional[int] = None    # Immediate value (enhanced tracking)
    instruction_binary: Optional[str] = None
    pre_rd: Optional[int] = None       # Pre-execution rd value
    pre_rs1: Optional[int] = None      # Pre-execution rs1 value
    pre_rs2: Optional[int] = None      # Pre-execution rs2 value
    actual_rd: Optional[int] = None    # Actual result rd value
    actual_rs1: Optional[int] = None   # Actual result rs1 value
    actual_rs2: Optional[int] = None   # Actual result rs2 value
    expected_rd: Optional[int] = None  # Expected rd result
    success: bool = False             # Test passed (updated logic)
    overflow: bool = False            # Overflow occurred
    illegal_instruction: bool = False  # Illegal instruction detected
    raw_text: str = ""                # Original log text
```

CocotbLogParser Class

Key Methods

- `parse_log_file(filename)`: Parse a complete log file
- `print_summary()`: Display comprehensive analysis summary (includes immediate statistics)
- `get_failed_tests()`: Return list of genuinely failed TestCase objects
- `get_instruction_analysis(instruction_name)`: Detailed analysis including immediate values
- `export_to_json(filename)`: Export all data to JSON format

Statistics Dictionary

The parser maintains comprehensive statistics in `self.stats`:

python

```
{
    'total_tests': int,
    'successes': int,                  # Includes successful overflows
    'failures': int,                   # Only genuine failures
    'overflows': int,                  # All overflows (successful and failed)
    'illegal_instructions': int,
    'instruction_counts': Counter,       # Instruction frequency
    'instruction_type_counts': Counter,   # R-Type, I-Type, etc.
    'register_usage': Counter,           # All register usage
    'rd_usage': Counter,                 # Destination register usage
    'rs1_usage': Counter,                # Source register 1 usage
    'rs2_usage': Counter,                # Source register 2 usage
    'immediate_values': Counter,         # Specific immediate values (enhanced)
    'immediate_ranges': Counter,         # Immediate value ranges (enhanced)
    'register_value_ranges': Counter,    # Register value ranges
    'pre_register_values': Counter,      # Pre-execution values
    'actual_register_values': Counter    # Result values
}
```

Integration with Testbench

Recommended Testbench Output Format

To maximize parser effectiveness, format your cocotb testbench output as follows:

```
python
```

```
# In your cocotb test
def log_test_result(test_num, instr_type, instr_name, registers,
                    binary, pre_vals, actual_vals, expected, success, overflow=False):
    print(f"Test {test_num}")
    print(f"Instruction type: {instr_type}")
    print(f"Name: {instr_name}")

    if 'imm' in registers:
        print(f"Registers: rd={registers['rd']}, rs1={registers['rs1']} imm={registers['imm']}")
        # Alternative formats also supported:
        # print(f"immediate: {registers['imm']}")
    else:
        print(f"Registers: rd={registers['rd']}, rs1={registers['rs1']} rs2={registers['rs2']}")

    print(f"Instruction: {binary}")
    print(f"Pre-instruction: rd={pre_vals['rd']}, rs1={pre_vals['rs1']}, rs2={pre_vals['rs2']}")
    print(f"Actual: rd={actual_vals['rd']}, rs1={actual_vals['rs1']}, rs2={actual_vals['rs2']}")

    if overflow:
        print(f"Expected rd: overflow")
    else:
        print(f"Expected rd: {expected}")

    if success:
        print("Success!")
    else:
        print("FAILED!")
    print()  # Blank line separator
```

Status Indicators (Updated)

The parser recognizes these status patterns with improved logic:

- **Success**: `Success!` in the log OR expected overflow matches actual OR expected value matches actual
- **Overflow**: `overflow` (case insensitive) or `Expected rd: overflow`
- **Illegal Instruction**: `illegal` (case insensitive)

Critical: Overflow is no longer automatically treated as failure. Tests expecting overflow that actually overflow are marked as successful.

Use Cases

Verification Coverage Analysis

- Track which instructions are being tested most frequently
- Identify untested or under-tested instructions
- Analyze register and immediate value usage patterns for coverage gaps

Test Quality Assessment

- Monitor success rates across different instruction types with proper overflow handling
- Identify problematic instructions or value ranges
- Track overflow behavior patterns (both expected and unexpected)

Debug Assistance

- Quickly identify genuinely failed tests (excluding expected overflows)
- Analyze failure patterns by instruction type
- Export data for external analysis tools

Immediate Value Analysis (New)

- Understand immediate value distribution in your tests
- Identify immediate value ranges that need more coverage
- Analyze patterns in immediate operand usage

Performance Analysis

- Understand test data distribution including immediate values
- Optimize test generation for better coverage
- Identify edge cases and boundary conditions

JSON Export Format

The exported JSON contains two main sections with enhanced immediate tracking:

```
json
```

```
{
  "stats": {
    "total_tests": 10000,
    "successes": 9998,
    "failures": 2,
    "overflows": 45,
    "instruction_counts": {"add": 543, "sub": 421, "addi": 234, ...},
    "register_usage": {"0": 123, "1": 456, ...},
    "immediate_values": {"0": 45, "1": 23, "-1": 67, "2048": 12, ...},
    "immediate_ranges": {"zero": 45, "±2^4 (pos)": 123, "±2^12 (neg)": 67, ...},
    ...
  },
  "test_cases": [
    {
      "test_number": 1,
      "instruction_type": "I-Type",
      "instruction_name": "addi",
      "rd": 20,
      "rs1": 8,
      "immediate": 42,
      "success": true,
      "overflow": false,
      ...
    },
    ...
  ]
}
```

Troubleshooting

Common Issues

1. **No tests parsed**: Check log format matches expected pattern
2. **Missing instruction data**: Ensure all required fields are present
3. **Incorrect success/failure counts**:
   - Verify `Success!` indicators in logs
   - Check that expected overflows are properly marked
   - Ensure `Expected rd: overflow` format for overflow tests
4. **Missing immediate values**:
   - Try multiple format variants (`imm=X`, `immediate: X`, etc.)

  - Check case sensitivity
  - Verify immediate values appear in expected locations

Overflow Handling Issues

If overflow tests are incorrectly marked as failures:

1. Ensure overflow tests use `Expected rd: overflow` format
2. Check that overflow indicator appears in the log text
3. Verify the test actually produces the expected overflow condition

Immediate Value Tracking Issues

If immediate values aren't being tracked:

1. Check the format matches one of the supported patterns
2. Try adding debug output to see what the parser is finding
3. Use `--export` to examine parsed test cases

Log Format Debugging

Use the `--export` option to examine how the parser interpreted your logs:

```bash
python3 parser.py testbench.log --export debug.json
# Examine debug.json to see parsed test cases and their immediate values
```

Recent Updates

Version 2.0 Changes

- **Fixed Overflow Logic**: Overflows no longer automatically count as failures
- **Enhanced Immediate Tracking**: Support for multiple immediate value formats
- **Improved Success Detection**: Better logic for determining test success
- **Extended Statistics**: Added comprehensive immediate value analysis
- **Better Analysis**: Instruction analysis now includes immediate value patterns

Contributing

To extend the parser:

1. **Add new status indicators**: Modify regex patterns in `parse_test_case()`
2. **Add immediate formats**: Add new patterns to the `imm_patterns` list
3. **Add value categories**: Extend `categorize_value()` method
4. **Add statistics**: Update `stats` dictionary and `update_stats()` method
5. **Add output formats**: Create new export methods

License

This tool is provided as-is for educational and research purposes. Feel free to modify and extend for your verification needs.