Exception Handling Module Documentation

Introduction

The Exception Handling module provides a comprehensive error management system for the RISC-V processor simulation. It includes custom exception classes and a centralized error handler that can operate in either strict mode (raising exceptions) or graceful recovery mode (handling errors silently with default values).

Key features:

- Custom exception hierarchy for different error types
- Centralized error handler with configurable strict/graceful modes
- Error logging and statistics tracking
- Enhanced memory classes with built-in error handling
- Support for program counter context in error messages

Class Structure

Exception Classes

1. `ProcessorException`: Base class for all processor exceptions
2. `MemoryException`: Memory access violations
3. `InvalidInstructionException`: Unknown or invalid instructions
4. `RegisterException`: Invalid register accesses
5. `ExecutionException`: General execution errors

Utility Classes

1. `ProcessorErrorHandler`: Central error handling with logging
2. `EnhancedDataMemory`: Memory with bounds checking
3. `EnhancedInstructionMemory`: Instruction memory with validation

Detailed Method Documentation

Exception Classes

`ProcessorException(message: str, pc: int = None, instruction: int = None)`

Base exception class with these features:

- Stores error message
- Optional program counter (PC) context
- Optional instruction value
- Human-readable string representation

Example:

```
raise ProcessorException("Division by zero", pc=0x100, instruction=0x1234)
```

Specialized Exceptions (all inherit from ProcessorException)

1. `MemoryException`: Memory access errors
2. `InvalidInstructionException`: Invalid opcodes
3. `RegisterException`: Invalid register numbers
4. `ExecutionException`: Other runtime errors

`ProcessorErrorHandler`

`__init__(strict_mode=False)`

Purpose: Initializes error handler

Parameters:

- `strict_mode`: If True, raises exceptions. If False, recovers gracefully.

Error Handling Methods

`handle_memory_error(address, operation, pc=None)`

Handles: Memory read/write errors

Behavior:

- Strict mode: Raises MemoryException
- Graceful mode: Returns 0 for reads, False for writes

`handle_invalid_instruction(instruction, pc=None)`

Handles: Unknown instructions

Behavior:

- Strict mode: Raises InvalidInstructionException
- Graceful mode: Treats as NOP (returns None)

`handle_register_error(reg_num, operation, pc=None)`

Handles: Invalid register accesses

Behavior:

- Strict mode: Raises RegisterException
- Graceful mode: Ignores operation (returns False)

`handle_execution_error(message, pc=None, instruction=None)`

Handles: General execution errors

Behavior:

- Strict mode: Raises ExecutionException
- Graceful mode: Continues execution (returns None)

Logging and Reporting

`_log_error(error_type, message, pc=None)`

Internal method: Records errors in log (keeps last 10)

`get_error_summary()`

Returns: Formatted string with error statistics

`reset_errors()`

Clears error log and counters

Enhanced Memory Classes

`EnhancedDataMemory`

`__init__(size=1024, base_address=0x1000, error_handler=None)`

Creates protected memory space

Key Methods:

- `read_word(address, pc=None)`: Safe memory read
- `write_word(address, value, pc=None)`: Safe memory write
- `_is_valid_address(address)`: Bounds checking

`EnhancedInstructionMemory`

`__init__(size=1024, error_handler=None)`

Creates protected instruction memory

Key Methods:

- `read_instruction(address)`: Safe instruction fetch
- `load_program(instructions, start_address=0)`: Validated program loading

Example Usage

```
# Create error handler (graceful mode by default)
error_handler = ProcessorErrorHandler()


# Create protected memory
memory = EnhancedDataMemory(size=1024, error_handler=error_handler)


# This will log error but not raise exception in graceful mode
value = memory.read_word(0x2000)  # Returns 0
print(value)  # 0


# Switch to strict mode
error_handler.strict_mode = True
try:
    memory.read_word(0x2000)  # Raises MemoryException
except MemoryException as e:
    print(f"Caught: {e}")


# View error log
print(error_handler.get_error_summary())
```

Key Error Cases Handled

1. Memory Errors:
    - Reading/writing outside allocated memory

    - Invalid memory addresses
2. Instruction Errors:
    - Unknown opcodes
    - Malformed instructions
3. Register Errors:
    - Accessing non-existent registers
    - Invalid register numbers
4. Execution Errors:
    - Arithmetic overflows
    - Invalid program loading
    - Other runtime issues

Testing

The module includes a test function (`test_error_handling()`) that demonstrates:

1. Strict vs graceful modes
2. Multiple error scenarios
3. Error logging
4. Protected memory operations

Run tests with:

```
if __name__ == "__main__":
    test_error_handling()
```

This exception handling system provides robust error management for the processor simulation while allowing flexible configuration of how errors should be handled (fail-fast vs fault-tolerant operation).