CLAUDE.md

This file provides guidance to Claude Code (claude.ai/code) when working with code in this repository.

Project Overview

68PHPhousand is a Motorola 68000 CPU emulator written in PHP. The design separates the CPU from external devices using dependency injection, with the CPU class implementing a trait-based architecture for better code organization despite internal coupling for performance.

Namespace: `ABadCafe\G8PHPhousand`

PHP Requirements: PHP 7.4+ with assertions enabled

Common Commands

Bootstrap/Class Map Management

The project uses a custom class map autoloader instead of Composer. When adding new classes, interfaces, or traits:

```
./updateclassmap
```

This regenerates `src/bootstrap.php` with the updated class map.

Running Tests

All tests require assertions enabled (`-dzend.assertions=1`):

```
# Run all unit tests
cd test && ./run_tests.sh

# Run Tom Harte test suite (requires test data)
cd test && ./run_harte.sh

# Run individual test files
php -dzend.assertions=1 test/test_memory.php
php -dzend.assertions=1 test/test_regs.php
php -dzend.assertions=1 test/test_eamodes.php
php -dzend.assertions=1 test/test_op_direct.php
php -dzend.assertions=1 test/test_scc.php
```

Note: The Tom Harte test suite requires downloading compressed JSON test cases from SingleStepTests/ProcessorTests into the `test/TomHarte/680x0/` directory.

Running Benchmarks

```
cd test
php bench_mem.php
php bench_rom.php
php bench_dbcc.php
php bench_dbf.php
```

Debugging

```
# Dump the opcode handler map
php test/dump_opcode_map.php
```

Architecture Overview

CPU Implementation

The CPU is implemented through trait composition in `Processor\Base`:

- **TRegisterUnit**: Data/Address register sets, Program Counter, Status Register, Condition Codes
- **TAddressUnit**: Effective Address (EA) mode implementations and management
- **TCache**: Optional caching for jump targets and immediate values
- **TArithmeticLogicUnit**: ALU operations and condition code calculations

- **TOpcode**: Opcode handler dispatching (exact and prefix-based)

Opcode implementation traits:

- **Opcode\TMove**: Move operations (MOVE, MOVEA, MOVEM, MOVEQ, LEA, PEA, EXG, SWAP)
- **Opcode\TLogical**: Logical operations (AND, OR, EOR, NOT) and shifts/rotates (ASL, LSL, ROL, ROXL, etc.)
- **Opcode\TSingleBit**: Bit manipulation (BTST, BSET, BCLR, BCHG)
- **Opcode\TArithmetic**: Arithmetic operations (ADD, SUB, ADDA, SUBA, ADDQ, SUBQ)
- **Opcode\TComparisonArithmetic**: Comparison operations (CMP, CMPA, CMPM)
- **Opcode\TBCDArithmetic**: BCD arithmetic (ABCD, SBCD, NBCD)
- **Opcode\TExtendedArithmetic**: Extended arithmetic (ADDX, SUBX, MULS, MULU, DIVS, DIVU)
- **Opcode\TShifter**: Memory shift/rotate operations
- **Opcode\TFlow**: Control flow (Bcc, DBcc, Scc, JMP, JSR, RTS, BSR)
- **Opcode\TSpecial**: Special operations (NOP, TRAP, RTE, RESET, STOP, etc.)

Opcode Handler System

The CPU uses two dispatch mechanisms:

1. **Exact Handlers** (`aExactHandler`): Array mapping complete 16-bit opcode values to closures for opcodes with no embedded parameters
2. **Prefix Handlers** (`aPrefixHandler`): Array mapping masked opcode prefixes to closures that extract parameters from the opcode word

Handlers are generated at CPU instantiation through:

- Direct closure definitions for simple operations
- Template-based metaprogramming via `Opcode\Template\TGenerator` for parameterized operations

Template System

Templates in `src/templates/operation/` generate opcode handlers at runtime:

- Templates are PHP files that output closure definitions as strings
- `Params` objects pass opcode information and cache flags to templates
- Generated code is evaluated into closures using `eval()` within an execution namespace

- Compiled handlers are cached by SHA-1 hash to avoid regeneration

Example template locations:

- `src/templates/operation/Bcc/*.tpl.php` - Branch instructions
- `src/templates/operation/DBcc/*.tpl.php` - Decrement and branch
- `src/templates/operation/Scc/*.tpl.php` - Set according to condition
- `src/templates/operation/arithmetic/*.tpl.php` - Arithmetic operations
- `src/templates/operation/logic/*.tpl.php` - Logical operations
- `src/templates/operation/move/*.tpl.php` - Move operations
- `src/templates/operation/bit/*.tpl.php` - Bit operations

Effective Address (EA) Modes

EA modes implement `EAMode\IReadOnly` (for source operands) or `EAMode\IReadWrite` (for destination operands):

Direct modes (register/immediate):

- `EAMode\Direct\DataRegister` - Data register direct (dN)
- `EAMode\Direct\AddressRegister` - Address register direct (aN)
- `EAMode\Direct\Immediate` - Immediate data (#N)

Indirect modes (memory access):

- `EAMode\Indirect\Basic` - Address register indirect (aN)
- `EAMode\Indirect\PostIncrement` - Post-increment (aN)+
- `EAMode\Indirect\PostIncrementSP` - Special case for A7
- `EAMode\Indirect\PreDecrement` - Pre-decrement -(aN)
- `EAMode\Indirect\PreDecrementSP` - Special case for A7
- `EAMode\Indirect\Displacement` - With displacement d16(aN)
- `EAMode\Indirect\Indexed` - With index d8(aN,xN.w|.l)
- `EAMode\Indirect\AbsoluteShort` - Absolute short address
- `EAMode\Indirect\AbsoluteLong` - Absolute long address
- `EAMode\Indirect\PCDisplacement` - PC relative with displacement
- `EAMode\Indirect\PCIndexed` - PC relative with index

EA mode traits:

- **TWithBusAccess**: Provides memory read/write via the bus interface

- **TWithExtensionWords**: Fetches extension words for addressing calculations
- **TWithLatch**: Caches calculated effective address for read-modify-write operations
- **TWithoutLatch**: No-op latch for EA modes that don't need it

Memory/Device System

All devices implement `Device\IBus` (union of `IDevice`, `IReadable`, `IWriteable`):

Memory implementations:

- `Device\Memory\BinaryRAM` - Contiguous binary string-backed RAM (big-endian)
- `Device\Memory\SparseRAM` - Associative array-backed RAM (byte-granular)
- `Device\Memory\SparseWordRAM` - Associative array-backed RAM (word-granular, fastest for code)
- `Device\Memory\SparseRAM24` - 24-bit address space variant
- `Device\Memory\CodeROM` - Read-only memory initialized from binary image

Other devices:

- `Device\NullDevice` - Ignores writes, returns zero for reads
- `Device\PageMap` - Memory mapping/banking support

All memory operations use unsigned values. Alignment is checked by the CPU, not memory implementations.

Test Harness

`TestHarness\CPU` extends `Processor\Base` with debugging capabilities:

- `executeAt()` - Execute single instruction at address
- `execute()` - Execute until exception/halt
- `executeTimed()` - Execute with timing information
- `dumpMachineState()` - Display register/memory state

`TestHarness\TomHarte` implements the Tom Harte test suite runner for validation against known-good test cases.

Key Implementation Details

Assertions

The codebase relies heavily on `assert()` for validation:

- Assertions MUST be enabled for tests (`zend.assertions=1`)
- Assertions should be disabled in production for performance (`zend.assertions=0`)
- Custom `AssertionFailureException` used in test harness

Performance Considerations

- CPU internals are tightly coupled for performance (avoiding indirection)
- Optional caching for jump targets and immediate values (enable via constructor parameter)
- `SparseWordRAM` provides fastest performance for code execution
- Template code generation eliminates runtime branching in opcode handlers

Memory Layout

- 68000 uses big-endian byte ordering
- Word (16-bit) and long (32-bit) accesses must be word-aligned
- Stack pointer (A7) maintains word alignment in pre-decrement/post-increment modes
- Address space is 24-bit (16MB) though some memory implementations support 32-bit

Condition Codes

Condition codes follow standard 68000 definitions (see `Processor\IConditionCode`):

- T/F (true/false)
- HI/LS (higher/lower or same)
- CC/CS (carry clear/set)
- NE/EQ (not equal/equal)
- VC/VS (overflow clear/set)
- PL/MI (plus/minus)
- GE/LT (greater or equal/less than)
- GT/LE (greater than/less or equal)

Working with Code

Adding New Opcodes

1. Determine if the opcode needs template generation or direct implementation

2. For templates: Create `.tpl.php` file in appropriate `src/templates/operation/` subdirectory
3. For direct: Add closure to appropriate trait in `src/Processor/Opcode/`
4. Register handler in corresponding `init*Handlers()` method
5. Run `./updateclassmap` if new files were added
6. Test with assertions enabled

Adding New EA Modes

1. Create class implementing `EAMode\IReadOnly` or `EAMode\IReadWrite`
2. Use appropriate traits (`TWithBusAccess`, `TWithExtensionWords`, etc.)
3. Implement `get{Byte,Word,Long}()` and optionally `set{Byte,Word,Long}()`
4. Register in `TAddressUnit::initEAModes()`
5. Run `./updateclassmap`

Adding New Memory Types

1. Implement `Device\IBus` interface
2. Implement `softReset()` and `hardReset()` from `IDevice`
3. Implement `read{Byte,Word,Long}()` and `write{Byte,Word,Long}()` methods
4. Ensure big-endian semantics for word/long operations
5. Run `./updateclassmap`

Troubleshooting

- **"Requires at least PHP 7.4"**: Upgrade PHP version
- **"Assertions must be enabled"**: Run with `-dzend.assertions=1`
- **"Unhandled Opcode"**: Opcode not registered in handler maps
- **Class not found**: Run `./updateclassmap` to regenerate autoloader
- **Template errors**: Check template syntax and `Params` object structure