opForge Reference Manual

This document describes the opForge assembler language, directives, and tooling. It follows a chapter layout similar to 64tass. Sections marked Planned describe features that are not implemented yet. This manual is validated against opForge CLI `0.9` (crate `0.9.0`). Release notes use feature-series labels (for example `v3.1` for the linker-region milestone).

1. Introduction

opForge is a two-pass, multi-CPU assembler for Intel 8080/8085, Z80, MOS 6502, and WDC 65C02 code. It supports:

- Dot-prefixed directives and conditionals.
- A 64tass-inspired expression syntax (operators, precedence, ternary).
- Preprocessor directives for includes and conditional compilation.
- Macro expansion with `.macro` and `.segment`.
- Optional listing, Intel HEX, and binary outputs.

The `.cpu` directive currently accepts `8080` (alias for `8085`), `8085`, `z80`, `6502`, `m6502`, and `65c02`.

There is no `.dialect` directive; dialect selection follows the CPU default.

2. Usage tips

- Directives, preprocessor directives, and conditionals are dot-prefixed (`.org`, `.if`, `.ifdef`).
- # prefixes immediate operands (for example `LDA #$10`).
- Macro invocation is dot-prefixed (for example `.COPY src,dst`).
- Labels may end with `:` or omit it.
- The program counter can be set with `* = expr` or `.org expr`.
- If no outputs are specified for a single input, the assembler defaults to list+hex when a root-module output name (or `-o`) is available.

3. Expressions and data types

3.1 Integers

Integer literals are supported in several formats:

- Decimal: `123`
- Hex: `$1234` or `1234h`
- Binary: `%1010` or `1010b`
- Octal: `17o` or `17q`

Underscores are allowed for readability (`$12_34`).

`$` evaluates to the current address.

3.2 Strings

Strings are quoted with `'` or `"` and are usable in data directives:

```
.byte "HELLO", 0
```

3.3 Booleans

Logical operators treat non-zero as true. Logical operators return `0` or `1`.

3.4 Symbols

Symbols are names bound to values. A symbol can be defined by a label (current address) or an assignment.

3.5 Expressions

Expressions are used in directives and operands. Unary < and > select the low or high byte of a value:

```
<($1234+1)
>($1234+1)
```

3.6 Expression operators

Operator set (highest level summary):

```
**  *  /  %  +  -  <<  >>  ==  !=  <>  <  <=  >  >=  &  ^  |  &&  ^^  ||  !  ~
?:
```

Concatenation uses `..`.

3.7 Conditional operator

The ternary operator `?:` is supported with standard precedence rules:

```
flag ? value_if_true : value_if_false
```

3.8 Planned data types

Planned (not implemented yet): bit strings, floating-point values, lists, tuples, code blocks, and type values.

4. Compiler directives

4.1 Assembling source

```
.include "file"
```

Notes:

- `.include` is literal text inclusion only; it does not participate in module loading.

4.2 Controlling the program counter

```
.org $1000
* = $2000
```

Section-local emission and linker-region placement:

```
.region ram, $1000, $10ff, align=16
.section data, kind=data, align=2, region=ram
.endsection
.place data in ram
.pack in ram : code, data
.align 16
```

Notes:

- `.section` selects a named emission target; `.endsection` restores the previous target.
- `.section` supports `kind=code|data|bss`, `align=<n>`, and `region=<name>`.
- `.place`/`.pack` assign final section base addresses via regions.

- `.dsection` is not supported and emits an error.
- `.org` and `.align` apply to the current emission target.
- Sections referenced by `.output` must be explicitly placed.
- `.mapfile` and `.exportsections` may include unplaced sections.

Linker output directives:

```
.output "build/out.bin", format=bin, sections=code,data
.output "build/image.bin", format=bin, image="$8000..$80ff", fill=$ff, contiguous=false, sections
.mapfile "build/out.map", symbols=all
.exportsections dir="build/sections", format=bin, include=bss
```

Output mode rules:

- Default output mode is contiguous (`contiguous=true`): selected sections must be adjacent.
- Image output mode (`image=... + fill=...`) allows sparse placement within the configured span.
- PRG output prefixes a 2-byte little-endian load address (`loadaddr=` optional).

4.3 Data directives

```
.byte expr[, expr...]
.db expr[, expr...]          ; alias for .byte
.word expr[, expr...]
.dw expr[, expr...]          ; alias for .word
.long expr[, expr...]
.ds expr
.emit unit, expr[, expr...]  ; unit = byte|word|long or numeric size
.res unit, count             ; BSS-only reservation
.fill unit, count, value     ; repeated data fill
```

Notes:

- `.emit` and `.fill` are data-emitting directives and are not allowed in `kind=bss` sections.
- `.res` is only allowed in `kind=bss` sections.
- For `word`, unit size follows current CPU word size.

4.4 Symbols and assignments

```
WIDTH = 40          ; read-only constant
var1  := 1          ; read/write variable
var2  :?= 5         ; only if undefined
var1  += 1          ; compound assignment
```

Compound assignment operators:

```
+= -= *= /= %= **= |= ^= &= ||= &&= <<= >>= ..= <?= >?= x= .=
```

`.const` and `.var` mirror = and := semantics; `.set` is an alias for `.var`.

4.5 Conditional assembly

```
.if expr
.elseif expr
.else
.endif
```

4.10 Modules and metadata

Modules define semantic scopes and imports; `.use` loads dependencies by module-id:

```
.module app.main
    .use util.math
    .pub
sum .const 0
.endmodule
```

Root-module metadata controls output naming:

```
.module main
    .meta
        .name "Demo Project"
        .version "1.0.0"
        .output
            .name "demo"
            .list
            .hex "demo-hex"
            .bin "0000:ffff"
            .fill "ff"
            .z80
                .name "demo-z80"
                .bin "0000:7fff"
                .fill "00"
            .endz80
        .endoutput
    .endmeta
    .meta.output.name "demo"
    .meta.output.z80.name "demo-z80"
.endmodule
```

Inside a `.meta` block, `.name` sets the metadata name. Inside an `.output` block (or `.meta.output.*` inline), `.name` sets the output base name. `.list`/`.hex`/`.bin`/`.fill` are valid only inside `.output` blocks or via `.meta.output.*` inline directives.

`.use` forms (module scope only):

```
.use util.math
.use util.math as M
.use util.math (add16, sub16 as sub)
.use util.math with (FEATURE=1, MODE="fast")
```

Notes:

- `.use` must appear inside a module and at module scope.
- `.use` affects runtime symbol resolution only.
- `.pub`/`.priv` visibility is enforced for runtime symbols (labels/constants/vars) only; macro/segment exports are not filtered by `.use`.

4.10.1 Root input

- `-i` accepts a file or folder.
- Folder input must contain exactly one `main.*` file (case-insensitive, `.asm` or `.inc`).
- The folder name becomes the input base (used for default output names).

4.10.2 Module identity

- If a file has no explicit `.module`, it defines an implicit module whose id is the file basename.
- If explicit modules exist, all top-level content must be inside `.module` blocks.
- The root module is:
  - the module matching the entry filename (case-insensitive), or
  - the first explicit module if no match exists.

4.10.3 Module resolution

- Search root: entry file directory only.
- Extensions: fixed to `.asm` and `.inc`.
- Module id matching is case-insensitive.
- If a file defines multiple modules, only the requested module is extracted.
- Missing or ambiguous module ids are errors; errors include an import stack.

4.10.4 Visibility rules

- `.pub`/`.priv` control runtime symbol visibility (labels/constants/vars).
- Macro/segment exports are not filtered by `.use`.

4.10.5 Root metadata output rules Output base precedence:

1. `-o`/`--outfile`
2. `.meta.output.<target>.name`
3. `.meta.output.name`
4. input base (file basename or folder name)

Examples in the repo:

- examples/module_use_autoload.asm
- examples/module_metadata_output.asm
- examples/project_root/main.asm

Match form:

```
.match expr
.case expr[, expr...]
    ; body
.case expr
    ; body
.default
    ; body
.endmatch
```

The match expression is evaluated once; the first matching `.case` wins, and `.default` is used if no case matches.

4.6 Scopes

Scopes are introduced by `.block` and closed by `.endblock`:

```
OUTER .block
INNER .block
VAL   .const 5
.endblock
.endblock
```

```
.word OUTER.INNER.VAL
```

Symbol lookup searches the current scope first, then parent scopes, then global.

4.7 Target CPU

```
.cpu 8080    ; alias for 8085
.cpu 8085
.cpu z80
.cpu 6502
.cpu m6502
.cpu 65c02
```

Planned (not currently supported): `65816`, `45gs02`, `68000` and related CPUs.

4.8 End of assembly

```
.end
```

4.9 Preprocessor directives

Preprocessor directives are dot-prefixed:

```
.ifdef NAME
.ifndef NAME
.elseif NAME
.else
.endif
.include "file"
```

Notes:

- `#` is used for immediate operands; preprocessor directives must use dot form.
- Preprocessor directives run before macro expansion.
- Preprocessor symbols are provided via the `-D/--define` command-line option.

5. Pseudo instructions

5.1 Macros

```
NAME .macro a, b=2
    .byte .a, .b
.endmacro
```

`.endm` is an alias for `.endmacro`.

Alternate directive-first form:

```
.macro NAME(a, b=2)
    .byte .a, .b
.endmacro
```

Invoke with `.NAME`:

Parenthesized call form:

```
.NAME(1)
```

5.2 Macro parameters

- Positional: `.1` .. `.9`
- Named: `.name` or `.{name}`
- Full argument list: `.@`
- Text form: `@1` .. `@9`

5.3 Segment macros

`.segment` defines a macro that expands inline without an implicit `.block`/`.endblock` wrapper:

```
INLINE .segment v
    .byte .v
.endsegment

.INLINE 7
```

Alternate directive-first form:

```
.segment INLINE(v)
    .byte .v
.endsegment
```

`.ends` is an alias for `.endsegment`.

5.4 Repetition

Planned (not implemented yet): repeat/loop-style directives.

5.5 Statement patterns (**`.statement`**)

`.statement` defines a patterned statement signature that is matched when the statement label appears without a leading dot:

```
.statement move.b char:dst "," char:src
    .byte 'b'
    .byte '.dst', 0
    .byte '.src', 0
.endstatement

move.b d0, d2
```

Rules:

- Typed captures use the explicit `type:name` form (e.g. `byte:val`, `char:reg`).
- Literal commas must be quoted as `","` inside signatures.
- Statement labels may include dots (e.g. `move.b`, `move.l`).
- Boundary spans `[{ ... }]` enforce adjacency rules within the span.

Capture types (built-in):

- `byte`, `word`, `char`, `str`
- `byte`/`word` enforce numeric literal range checks.
- Identifiers/registers also match capture types (resolved later by expression handling).

Expansion model:

- `.statement` definitions are expanded by the macro processor before parsing.
- Statement definitions are global (not module-scoped).

5.6 Assembler pipeline (CPU family/dialect)

Registry

- Families and CPUs are registered in Rust at startup.
- CPU names are case-insensitive and resolved via the registry.

Selection

- `.cpu <name>` selects the active CPU.
- There is no **`.dialect`** directive.
- Dialect mapping (when present) is selected by CPU default or family canonical dialect.

Encoding pipeline

1. Parse operands with the family handler.
2. Apply dialect mapping (mnemonic + operand rewrite).
3. Attempt family pre-encode (optional).
4. Resolve operands with the CPU handler.
5. Run CPU validator (trait exists, currently unused by CPUs).
6. Encode with family handler; fall back to CPU handler for extensions.

6. Compatibility

- Dot-prefixed directives are required (for `.org`, `.set`, `.if`, etc.).
- Labels may omit the trailing `:`.

7. Command line options

Syntax:

`opForge [OPTIONS]`

Inputs (required):

- `-i, --infile <FILE|FOLDER>`: input `.asm` file or folder (repeatable). Folder inputs must contain exactly one `main.*` root module.

Outputs:

- `-l, --list [FILE]`: listing output (optional filename).
- `-x, --hex [FILE]`: Intel HEX output (optional filename).
- `-b, --bin [FILE:ssss:eeee|ssss:eeee|FILE]`: binary image with optional range(s), repeatable.

Other options:

- `-o, --outfile <BASE>`: output base name if output filename omitted.
- `-f, --fill <hh>`: fill byte for binary output (hex). Requires binary output. Defaults to `FF`.
- `-g, --go <aaaa>`: execution start address in HEX output. Requires HEX output.
- `-D, --define <NAME[=VAL]>`: predefine macro (repeatable).
- `-c, --cond-debug`: include conditional state in listing.
- `--pp-macro-depth <N>`: maximum preprocessor macro expansion depth (default 64, minimum 1).
- `-h, --help`: print help.
- `-V, --version`: print version.

Notes:

- If multiple inputs are provided, `-o` must be a directory and explicit output filenames are not allowed; each input uses its own base name under the output directory.

- With multiple inputs, at least one output type (−l, −x, −b) must be selected.
- If no outputs are specified for a single input, opForge defaults to list+hex when `.meta.output.name` (or −o) is available.
- −b without a range emits a binary that spans the emitted output.

8. Messages

Diagnostics include a line/column and a highlighted span in listings. Terminal output may use ANSI colors to highlight the offending region.

Listing addresses reflect the current emission context: inside a `.section` the address column shows the section-local program counter. Absolute placed output is shown in the generated-output footer table.

Common linker-region failures:

- `.dsection has been removed; use .place/.pack with .output`
- `Section referenced by .output must be explicitly placed`
- `contiguous output requires adjacent sections` (gap diagnostic includes range)

9. Credits

opForge is derived from the asm85 assembler by Tom Nisbet and has been extended with new expression syntax, directives, and tooling.

10. Default translation

Planned (not implemented yet): translation tables for character/byte mappings.

11. Escapes

Strings accept the following escapes:

```
\n  \r  \t  \0  \xHH
```

Any other escape sequence inserts the escaped character as-is.

12. Opcodes

Instruction mnemonics are selected by `.cpu`:

- Intel dialect for 8080/8085 (`MOV`, `MVI`, `JMP`, ...)
- 8085-only additions include `RIM` and `SIM`.
- Zilog dialect for Z80 (`LD`, `JP`, `JR`, ...), including `SLL` and half-index registers (`IXH`, `IXL`, `IYH`, `IYL`).
- Standard MOS 6502/65C02 mnemonics (`LDA`, `JMP`, `BRA`, ...), including 65C02 additions such as `STP`, `WAI`, `DEC A`/`INC A` (`DEA`/`INA` aliases), and extended `BIT` modes.

13. Appendix: quick reference

13.1 Directives

```
.org  .align  .region  .place  .pack  .section  .endsection  .cpu  .end
.byte  .db  .word  .dw  .long  .ds  .emit  .res  .fill
.const  .var  .set
.if  .elseif  .else  .endif  .match  .case  .default  .endmatch
.ifdef  .ifndef  .include
.module  .endmodule  .use  .pub  .priv  .block  .endblock
```

```
.macro  .endmacro  .endm  .segment  .endsegment  .ends  .statement  .endstatement
.meta .endmeta .name .version .output .endoutput .list .hex .bin .mapfile .exportsections
.meta.name  .meta.version
.meta.output.name .meta.output.<target>.name .meta.output.list .meta.output.hex .meta.output.b
```

13.2 Assignment operators

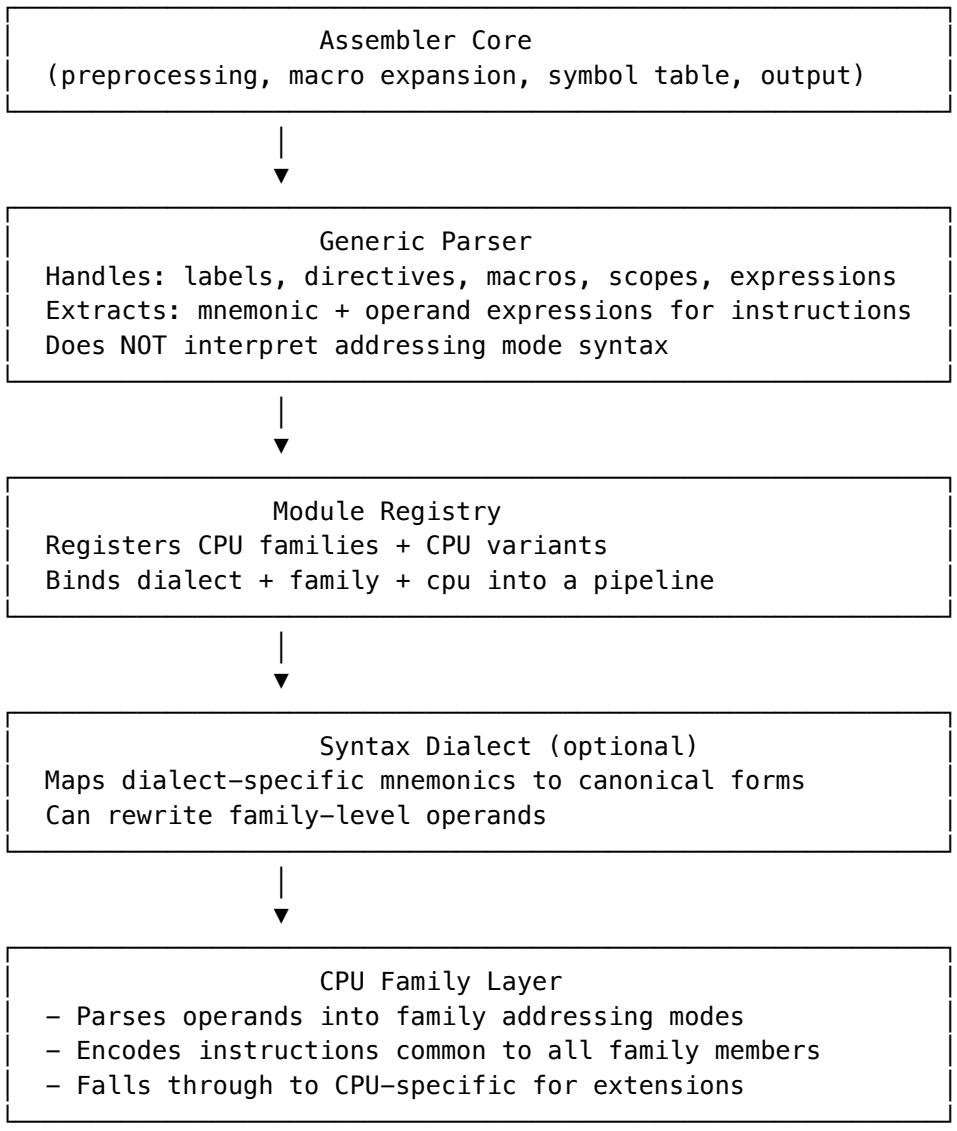
```
=  :=  :?=  +=  -=  *=  /=  %=  **=  |=  ^=  &=  ||=  &&=  <<=  >>=  ..=  <?=  >?=  x=  .=
```

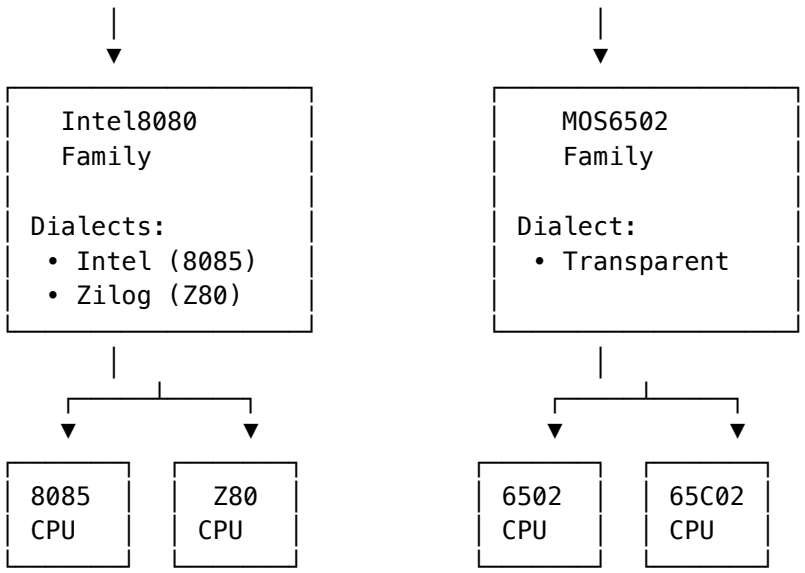
14. Appendix: multi-CPU architecture

This appendix describes the modular architecture that allows opForge to support multiple CPU targets (8085, Z80, 6502, 65C02) through a common framework.

Overview

The assembler is organized into layers with hierarchical parsing and encoding:





Layer responsibilities

Assembler Core

- File inclusion and preprocessing
- Output generation (listing files, hex files)
- Error collection and reporting
- Two-pass assembly coordination

Generic Parser

- Labels, directives, macros, scopes, expressions, comments
- Extracts instruction mnemonic + operand expressions
- Does not interpret addressing mode syntax

Module Registry

- Registers family and CPU handlers
- Resolves pipeline (family + CPU + dialect mapping)

Family Handler

- Parses family-common operand syntax
- Encodes family-common instructions
- Falls through to CPU handler for extensions

CPU Handler

- Resolves ambiguous operands
- Encodes CPU-specific instructions
- Validates CPU-specific constraints

Hierarchical processing

1. Generic parser extracts mnemonic + operand expressions.
2. Family handler parses expressions into family operands.
3. Dialect mapping rewrites mnemonic/operands (if needed).
4. Family pre-encode (optional) attempts encoding from family operands.
5. CPU handler resolves ambiguous operands to CPU-specific operands and applies CPU validation.

6. Encode with family handler first; CPU handler encodes extensions when family returns `NotFound`.

Family extensions

MOS 6502 Family

Operand syntax extensions:

Syntax	6502 (Base)	65C02 (Extended)
`#$20`	Immediate ✓	Immediate ✓
`$20`	Zero Page ✓	Zero Page ✓
`($20,X)`	Indexed Indirect ✓	Indexed Indirect ✓
`($20),Y`	Indirect Indexed ✓	Indirect Indexed ✓
`($20)`	✗ Invalid	Zero Page Indirect ✓
`($1234,X)`	✗ Invalid	Absolute Indexed Indirect ✓

Instruction extensions:

Instruction	6502 (Base)	65C02 (Extended)
`LDA`	✓ All modes	✓ All modes + ($zp)
`BRA`	✗	✓ Branch Always
`PHX`, `PLX`	✗	✓ Push/Pull X
`PHY`, `PLY`	✗	✓ Push/Pull Y
`STP`, `WAI`	✗	✓ Stop/Wait
`DEC A`/`INC A`	✗	✓ Accumulator mode (`DEA`/`INA` aliases)
`BIT #imm`, `BIT zp,X`, `BIT abs,X`	✗	✓ Extended BIT modes
`STZ`	✗	✓ Store Zero
`TRB`, `TSB`	✗	✓ Test and Reset/Set Bits
`BBSn`, `BBRn`	✗	✓ Branch on Bit Set/Reset
`RMBn`, `SMBn`	✗	✓ Reset/Set Memory Bit

Intel 8080 Family

Operand syntax extensions:

Syntax	8080/8085 (Base)	Z80 (Extended)
`A`, `B`, `HL`	Register ✓	Register ✓
`(HL)`	Indirect ✓	Indirect ✓
`(IX+d)`, `(IY+d)`	✗	CB-prefix targets only ✓ (general forms not yet encoded)

Instruction extensions:

The Z80 adds `DJNZ`, `JR` (with conditions), `EX`, `EXX`, block operations (`LDI`, `LDIR`, etc.), bit operations (`BIT`, `SET`, `RES`), and shifts/rotates including `SLL`. Z80-specific registers include `IX`, `IY`, `IXH`, `IXL`, `IYH`, and `IYL`.

Syntax dialects (8080 vs Z80)

Operation	8080/8085 Dialect	Z80 Dialect	Opcode
Move register	`MOV A,B`	`LD A,B`	78

Operation	8080/8085 Dialect	Z80 Dialect	Opcode
Move immediate	`MVI A,55h`	`LD A,55h`	3E 55
Load direct	`LDA 1234h`	`LD A,(1234h)`	3A 34 12
Store direct	`STA 1234h`	`LD (1234h),A`	32 34 12
Jump	`JMP 1000h`	`JP 1000h`	C3 00 10
Jump if zero	`JZ 1000h`	`JP Z,1000h`	CA 00 10
Call	`CALL 1000h`	`CALL 1000h`	CD 00 10
Return	`RET`	`RET`	C9
Add register	`ADD B`	`ADD A,B`	80
Add immediate	`ADI 10h`	`ADD A,10h`	C6 10

Core abstractions

- CpuType: concrete processor (I8085, Z80, M6502, M65C02)
- CpuFamily: processor family (Intel8080, MOS6502)

Handler traits (summary)

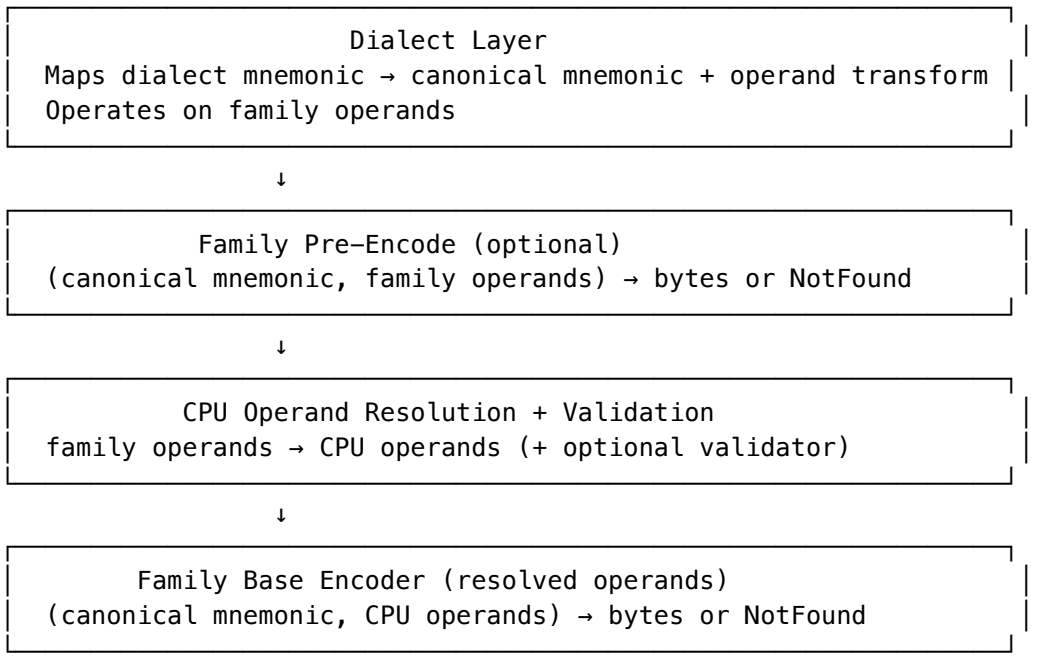
FamilyHandler provides:

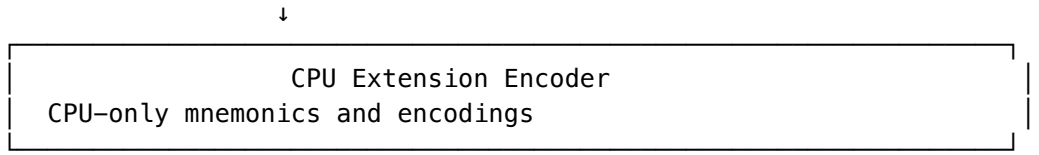
- Operand parsing for family-common syntax
- Optional pre-encoding using family operands
- Instruction encoding for family-common mnemonics
- Register and condition code recognition

CpuHandler provides:

- Resolution of ambiguous operands to CPU-specific forms
- Instruction encoding for CPU-specific mnemonics
- Query methods for supported mnemonics

Instruction resolution architecture





Module interfaces (summary)

```rust
pub trait FamilyModule: Send + Sync {
    fn family_id(&self) -> CpuFamily;
    fn family_cpu_id(&self) -> Option<CpuType> { None }
    fn family_cpu_name(&self) -> Option<&'static str> { None }
    fn cpu_names(&self, registry: &ModuleRegistry) -> Vec<String>;
    fn canonical_dialect(&self) -> &'static str;
    fn dialects(&self) -> Vec<Box<dyn DialectModule>>;
    fn handler(&self) -> Box<dyn FamilyHandlerDyn>;
}

pub trait CpuModule: Send + Sync {
    fn cpu_id(&self) -> CpuType;
    fn family_id(&self) -> CpuFamily;
    fn cpu_name(&self) -> &'static str;
    fn default_dialect(&self) -> &'static str;
    fn handler(&self) -> Box<dyn CpuHandlerDyn>;
    fn validator(&self) -> Option<Box<dyn CpuValidator>> { None }
}

pub trait DialectModule: Send + Sync {
    fn dialect_id(&self) -> &'static str;
    fn family_id(&self) -> CpuFamily;
    fn map_mnemonic(
        &self,
        mnemonic: &str,
        operands: &dyn FamilyOperandSet,
    ) -> Option<(String, Box<dyn FamilyOperandSet>)>;
}
```