Atlas Toolchain

Assembly & Linking

How the Atlas assembler and linker transform human-readable assembly source files into executable machine code — covering every stage of the pipeline, the binary file formats involved, and the mechanisms that make cross-file references work.

Jakob Flocke · Atlas Project · February 2026

Contents

Pipeline Overview

```
┌──────────┐          ┌──────────┐
│ main.asm │          │ math.asm │    source files
└────┬─────┘          └────┬─────┘
     │  atlas asm          │  atlas asm
     ▼                     ▼
┌──────────┐          ┌──────────┐
│ main.o   │          │ math.o   │    object files (ATOB)
└────┬─────┘          └────┬─────┘
     │                     │
     └──────────┬──────────┘
                │  atlas ld
                ▼
         ┌────────────┐
         │ output.bin │                flat binary  — or —
         │ output.hex │                Intel HEX
         └────────────┘
```

Each `.asm` file is assembled **independently** into an object file (`.o`). The object files are then fed to the linker, which merges them, resolves all cross-file symbol references, and writes the final executable image.

The Atlas Instruction Set (ISA)

Atlas is a **16-bit** architecture. Every instruction encodes into exactly one 16-bit word (2 bytes), stored **big-endian** in memory and in object files (high byte at lower address).

Instruction Types

Type	Bits [15:12]	Description	Operands
A	`0000`	ALU register–register	`dest (4b)`, `source (4b)`, `op (4b)`
I	`0001`-`0101`	Immediate (8-bit)	`dest (4b)`, `imm (8b)`
M	`0110`-`0111`	Memory load / store	`dest (4b)`, `base (4b)`, `offset (4b)`
BI	`1000`	Branch (immediate target)	`abs (1b)`, `cond (3b)`, `addr (8b)`
BR	`1001`	Branch (register pair target)	`abs (1b)`, `cond (3b)`, `hi (4b)`, `lo (4b)`
S	`1010`	Stack (push / pop / sp adjust)	`op (4b)`, `reg (4b)`
P	`1011`	Peek / Poke (I/O ports)	`op (1b)`, `reg (3b)`, `offset (8b)`
X	`1100`	Extended / system	`op (4b)`, `operand (8b)`

Encoding Details

A-type — The top nibble is `0`. The remaining 12 bits carry `dest[11:8]`, `source[7:4]`, and `op[3:0]`, where `op` selects one of the 16 ALU operations (ADD, SUB, AND, OR, CMP, MOV, etc.).

opcode `0000`	**dest**	**source**	**ALU op**

I-type — The top nibble is `1 + op` (LDI = 1, ADDI = 2, SUBI = 3, ANDI = 4, ORI = 5). Bits `[11:8]` are the destination register. Bits `[7:0]` hold an 8-bit unsigned immediate value (max 255 / `0xFF`).

opcode	**dest**	**immediate (8 bits)**

M-type — Load (`0110`) and Store (`0111`). Syntax: `ld rD, [rB, offset]`. The offset fits in 4 bits.

opcode	**dest**	**base**	**offset**

BI-type — Bit `[11]` selects absolute vs.
relative addressing. Bits `[10:8]` encode the condition code (unconditional, EQ, NE, CS, CC, MI, PL). Bits `[7:0]` hold the 8-bit branch target address.

`1000`	**abs**	**cond**	**address (8 bits)**

X-type — Top nibble `1100`. Bits `[11:8]` select the operation (HALT, SYSC, ERET, cache ops). The lower 8 bits carry optional operand data.

`1100`	**op**	**operand (8 bits)**

Registers

Atlas has 16 general-purpose registers (`r0`-`r15`), with conventions:

Register	Alias	Purpose
`r0`	—	Zero / scratch
`r10`	`tr`	Temporary register
`r12`	`sp`	Stack pointer
`r14`	`pc`	Program counter

Assembly Source Language

Basic Syntax

```
; This is a comment (semicolons to end of line)

label_name:               ; defines a label at the current address
    mnemonic operands     ; instruction (indentation is optional)
```

Directives

Directives start with a dot (`.`) and control the assembler's behaviour rather than producing instructions directly.

Directive	Syntax	Effect
`.global` / `.export`	`.global name`	Mark a symbol as globally visible for linking
`.import`	`.import name`	Declare a symbol defined in another file
`.imm`	`NAME: .imm value`	Define a named constant (not placed in memory)
`.text`	`.text`	Switch to the `.text` section (code)
`.data`	`.data`	Switch to the `.data` section
`.bss`	`.bss`	Switch to the `.bss` section (zero-initialised)
`.section`	`.section name`	Switch to an arbitrary named section
`.byte`	`.byte 0x41, 0x42`	Emit raw bytes into the current section
`.word`	`.word 0x1234`	Emit 16-bit words
`.ascii`	`.ascii "hello"`	Emit a string as raw bytes

Labels

A label is a name followed by a colon. It records the current byte offset within the current section:

```
loop:
    add r1, r2
    br loop            ; refers back to the address of `add`
```

Labels are **local** by default. To make a label visible to the linker (so other files can reference it), you must `.export` it:

```
.export my_function
my_function:
    ...
```

Named Constants (`.imm`)

A constant assigns a fixed numeric value to a name without placing anything in the output section. It uses the special syntax `NAME: .imm value`:

```
BUFFER_SIZE: .imm 64
IO_PORT:     .imm 0x80
```

Constants live in a virtual section called `.abs` (absolute). They are resolved at assemble time and substituted directly into instruction immediates.

Imports

When your code references a symbol defined in a different source file, you must declare it with `.import`:

```
.import add_values
    br add_values       ; will be resolved by the linker
```

Note

Without an `.import`, the assembler treats `add_values` as an undefined symbol and fails.

The Assembler — Stage by Stage

Assembly is a **two-pass** process implemented by the `atlas-assembler` crate.

Pass 1 — Lexing and Parsing

The source text is consumed token by token by the **Lexer**, which recognises:

- **Mnemonics** (`add`, `ldi`, `br`, `halt`, …)
- **Registers** (`r0`–`r15`, plus aliases like `sp`, `tr`, `pc`)
- **Immediates** (decimal, `0x` hex, `0b` binary)
- **Label definitions** (`name:`)
- **Label references** (bare `name` used as an operand)
- **Directives** (`.global`, `.import`, `.byte`, etc.)
- **Punctuation** (`,`, `[`, `]`, `@`)

The **Parser** then consumes the token stream and produces a flat list of `ParsedItem` values. Each item is one of:

Variant	Meaning
`Instruction(ParsedInstruction)`	A fully parsed instruction (opcode + operands)
`Data(Vec<u8>)`	Raw bytes from `.byte` / `.word` / `.ascii`
`SectionChange(String)`	The parser encountered a section directive

While parsing, the parser simultaneously populates a **symbol table** that tracks:

- **Labels** — name → (byte offset, section)
- **Constants** — name → value
- **Exports** — set of names marked `.global`
- **Imports** — set of names declared `.import`

Labels record the current byte position within their section at the point they are defined. This is why the parser maintains a running position counter (`pos`) that increments by 2 for every instruction and by the data length for `.byte` / `.word` / `.ascii`.

Pass 2 — Encoding

After all items are collected and the symbol table is complete, the assembler walks the item list and encodes each instruction into a 16-bit word.

Before encoding, it attempts **local resolution**: if an instruction references a label or constant that is defined in the *same* file, the assembler substitutes the resolved numeric value directly into the instruction's operand field. For example:

```
IO_PORT: .imm 0x80
    ldi r3, IO_PORT     ; resolved to: ldi r3, 0x80
```

The label `IO_PORT` is a constant with value `0x80`. During local resolution, the `Operand::Label("IO_PORT")` is replaced with `Operand::Immediate(0x80)` before encoding.

What happens with unresolved references?

If a label reference cannot be resolved locally (typically because it was declared via `.import`), the assembler:

1. **Substitutes a placeholder** — it replaces the label with `Immediate(0)`, producing a valid but incorrect encoding.
2. **Records a relocation** — it creates an `UnresolvedReference` noting the byte offset within the section, the section name, and the symbol name.

Key Insight

This allows encoding to succeed for every instruction, even when the final address is unknown. The linker will later patch these placeholders.

Encoding output

Each instruction is encoded into 2 bytes (big-endian) and appended to the current section's byte buffer. For a `.text` section containing 5 instructions, the buffer will be 10 bytes long.

Object File Emission

After encoding, the assembler constructs an `ObjectFile` struct containing:

Field	Source
`sections`	The byte buffers built during encoding, keyed by section name
`symbols`	Every label, constant, and import from the symbol table
`relocations`	Every unresolved reference that needs linker patching
`version`	Currently `1`

This struct is then serialised to disk in the **ATOB** binary format (described in the next section).

The Object File Format (`.o`)

Object files use a custom binary format identified by the magic bytes `ATOB` (Atlas Object Binary). All multi-byte integers are **little-endian**.

File Layout

```
┌───────────────────────────────┐
│  Magic: "ATOB" (4 bytes)      │
│  Version: u32                 │
│  Section count: u32           │
│  Symbol count: u32            │
│  Relocation count: u32        │
├───────────────────────────────┤
│  Section 0 … Section N        │
├───────────────────────────────┤
│  Symbol 0 … Symbol N          │
├───────────────────────────────┤
│  Relocation 0 … Relocation N  │
└───────────────────────────────┘
```

Section Record

Field	Description
`name_length: u32`	Length of the section name string
`name: [u8; name_length]`	UTF-8 string (e.g. `.text`)
`start: u32`	Start address (currently always 0)
`data_length: u32`	Length of the raw section content
`data: [u8; data_length]`	Encoded instructions / data bytes

Symbol Record

Field	Description
`name_length: u32`	Length of the symbol name
`name: [u8; name_length]`	UTF-8 symbol name
`value: u32`	Offset within section (or absolute value)

`has_section: u8`	1 = defined, 0 = undefined (import)
`section_length: u32` (if defined)	Length of the section name
`section: [u8; section_length]` (if defined)	e.g. `.text`, `.abs`
`binding: u8`	0 = Local, 1 = Global

- **Defined labels** have `has_section = 1` and their section set to whichever section they were defined in (usually `.text`). The value is the byte offset within that section.
- **Constants** (`.imm`) have `has_section = 1` with section `.abs`. The value is the constant's numeric value.
- **Imports** have `has_section = 0` and `binding = Global`. The value is 0 (meaningless until the linker resolves it).

Relocation Record

Field	Description
`offset: u32`	Byte offset within the section
`symbol_length: u32`	Length of the symbol name
`symbol: [u8; symbol_length]`	Name of the referenced symbol
`addend: i32`	Value to add after resolution (usually 0)
`section_length: u32`	Length of the section name
`section: [u8; section_length]`	Which section contains the reference

In plain English

Each relocation says: *"at byte `offset` within section `section`, there is a placeholder that should be replaced with the address of `symbol` + `addend`."*

Symbols in Detail

Symbol Kinds

Kind	Section	Binding	Created by
Local label	`.text` (or other)	Local	`label:`
Exported label	`.text` (or other)	Global	`label:` + `.export label`
Constant	`.abs`	Local (or Global)	`NAME: .imm value`
Import	None (undefined)	Global	`.import name`

Visibility and Binding

- **Local** symbols are visible only within the file that defines them. The linker sees them (they're stored in the `.o`) but won't use them to satisfy references from other files.
- **Global** symbols are visible across all files during linking. A global symbol may only be defined once; if two files both export the same name, the linker raises a **duplicate symbol** error.

How `.export` and `.import` Interact

Consider two files:

main.asm

```
.import add_values
.export main

main:
    ldi r1, 0x10
    br  add_values   ; → relocation
```

math.asm

```
.import return_here
.export add_values

add_values:
    add r1, r2
    br  return_here ; → relocation
```

After assembly:

- `main.o` contains a Global symbol `main` at offset 0 in `.text`, and an undefined Global symbol `add_values`.
- `math.o` contains a Global symbol `add_values` at offset 0 in `.text`, and an undefined Global symbol `return_here`.

The linker matches each file's undefined symbols against the other files' exported definitions.

Relocations in Detail

Why Relocations Exist

The assembler processes each file in isolation. When it encounters `br add_values` and `add_values` is declared `.import`, it has no idea what address that label will end up at — that depends on how the linker arranges all sections. So it:

1. Encodes the instruction with a **zero** in the immediate / address field.
2. Emits a relocation entry saying *"please patch offset X with the value of symbol Y"*.

What Gets Relocated

Only instructions with **label operands** that couldn't be resolved locally generate relocations:

- **I-type** instructions (`ldi`, `addi`, …) with a label in the immediate field
- **BI-type** instructions (`br`, `beq`, `bne`, …) with a label target
- **P-type** instructions (`peek`, `poke`) with a label offset

Remember

Constants (`.imm`) and local labels are resolved during assembly and do **not** generate relocations.

Relocation Fields

Field	Meaning
`offset`	Byte position within the section where the placeholder lives
`symbol`	Name of the symbol whose address should be substituted
`addend`	Signed integer added to the resolved address (usually 0)
`section`	Which section contains the instruction to patch

How the Linker Applies Relocations

When the linker processes a relocation:

1. It looks up `symbol` in the global symbol table to get the final address.
2. It computes `final_value = address + addend`.
3. It validates that `final_value` fits in the 8-bit immediate field (≤ `0xFF`).
4. It locates the instruction at `section_base + offset` in the merged section data.
5. It **keeps the upper byte** of the 16-bit instruction word (opcode, condition codes, register fields) and **replaces the lower byte** with `final_value`.

Why this works

All relocatable instruction types (I, BI, P) store their immediate/address in bits `[7:0]` — the low byte.

The Linker — Stage by Stage

The linker (`atlas-linker` crate) takes one or more `.o` files and produces a single flat executable image.

Stage 1 — Load Object Files

Each input `.o` file is parsed from the ATOB binary format back into an `ObjectFile` struct (sections, symbols, relocations).

Stage 2 — Merge Sections

Sections with the same name are **concatenated** in input order. For example, if `main.o` has a `.text` section of 20 bytes and `math.o` has a `.text` section of 4 bytes, the merged `.text` section will be 24 bytes with `main.o`'s code at offset 0 and `math.o`'s code at offset 20.

The linker tracks a **section base** for each (file, section) pair: the byte offset within the merged section where that file's contribution starts.

```
Merged .text:
┌──────────────────────────┬────────┐
│  main.o .text (20 bytes) │ math.o │
│  base = 0                │ base=20│
└──────────────────────────┴────────┘
```

Stage 3 — Build Global Symbol Table

The linker walks every symbol from every object file:

- **Undefined symbols** (imports, `section = None`) are skipped — they will be resolved when encountered as relocation targets.
- **Absolute constants** (section `.abs`) are registered at their literal value, without any base adjustment.
- **Defined labels** have their value adjusted by adding the section base for that file.

Error

If a global symbol is defined in two different files, the linker reports a **duplicate symbol error** and aborts.

Stage 4 — Apply Relocations

For every relocation in every object file:

1. Compute `patch_offset = section_base[file, section] + relocation.offset` — this is where the placeholder lives in the merged data.
2. Look up `relocation.symbol` in the global symbol table. If not found → **unresolved symbol error**.
3. Compute `final_value = symbol_address + relocation.addend`.
4. Validate `final_value` ≤ `0xFF` (8-bit immediate constraint).
5. Patch: read the 2-byte instruction at `patch_offset`, keep the high byte, write `final_value` as the low byte.

Stage 5 — Write Output

The merged sections are flattened into a single byte stream. The `.text` section is placed first, followed by any other sections (`.data`, `.bss`, etc.) in alphabetical order.

The output format is chosen by file extension:

Extension	Format	Content
`.bin` (or any other)	Raw binary	Byte stream written directly to disk
`.hex`	Intel HEX	Byte stream encoded as ASCII Intel HEX records

Output Formats

Raw Binary (`.bin`)

The simplest format: the merged section bytes are written directly to a file with no header, no metadata. The file's first byte corresponds to address `0x0000`.

Tip

To load this into a simulator or FPGA memory, you just need to know that instructions start at offset 0.

Intel HEX (`.hex`)

Intel HEX is an ASCII format widely supported by EPROM programmers, FPGA tools, and emulators. Each line (called a *record*) has the structure:

```
:LLAAAATT[DD...]CC
```

Field	Size	Meaning
`:`	1 char	Start code
`LL`	2 hex chars	Byte count of the data payload
`AAAA`	4 hex chars	16-bit start address of this record
`TT`	2 hex chars	Record type (00 = data, 01 = EOF)
`DD…`	$2 \times$ LL chars	Data bytes
`CC`	2 hex chars	Two's-complement checksum

The toolchain emits **Data records** (type 00) with up to 16 data bytes each, followed by a single **EOF record** (`:00000001FF`).

The checksum is computed as:

$$\mathrm{CC} = \left(\neg\left(\mathrm{LL} + \mathrm{AAAA}_{\mathrm{hi}} + \mathrm{AAAA}_{\mathrm{lo}} + \mathrm{TT} + \sum \mathrm{DD}_i\right) + 1\right) \wedge \mathrm{0xFF}$$

Example: the 2-byte instruction `0x1110` (`ldi r1, 0x10`) at address `0x0000` produces:

```
:020000001110DD
```

Where `0x02 + 0x00 + 0x00 + 0x00 + 0x11 + 0x10 = 0x23`, and `(!0x23 + 1) & 0xFF = 0xDD`.

Worked Example

The repository ships with a three-file test program in `test/` that exercises most of the toolchain's features.

Source Files

The program is split across three modules:

File	Exports pose	Imports
`main.asm`	`main`, `mul_ret`, `div_ret`, `abs_ret`, `io_ret` harness — runs 6 tests, reports pass/fail	`multiply`, `divide`, `abs_value`, `emit_byte`, `read_byte`
`math.asm`	`multiply`, `divide`, `abs_value` metic library (multiply, divide, abs)	`mul_ret`, `div_ret`, `abs_ret`
`io.asm`	`emit_byte`, `read_byte` O port routines (peek/poke)	`io_ret`

main.asm (abridged)

```
; External routines
.import multiply
.import divide
```

```
.import abs_value
.import emit_byte
.import read_byte

; Return-point labels
.export mul_ret
.export div_ret
.export abs_ret
.export io_ret
.export main

; Constants
STACK_TOP:    .imm 0xF0
RESULT_ADDR: .imm 0x80
MAGIC:        .imm 0xAA
NUM_TESTS:    .imm 0x06

main:
    ldi  sp, STACK_TOP         ; initialise stack
    ldi  r9, 0x00              ; test-pass counter

test_add:                      ; TEST 1: 0x10 + 0x25 = 0x35
    ldi  r1, 0x10
    ldi  r2, 0x25
    add  r1, r2
    ldi  r5, 0x35
    cmp  r1, r5
    bne  test_sub
    addi r9, 0x01

test_mul:                      ; TEST 5: cross-module 6 × 7 = 42
    ldi  r1, 0x06
    ldi  r2, 0x07
    br   multiply              ; ← RELOCATION (import)
mul_ret:                       ; multiply branches back here
    ldi  r5, 0x2A
    cmp  r1, r5
    bne  test_mem
    addi r9, 0x01

test_mem:                      ; TEST 6: memory round-trip
    ldi  r1, 0xBE
    ldi  r3, RESULT_ADDR       ; ← constant, resolved locally
    st   r1, [r3, 0]
    ldi  r1, 0x00
    ld   r1, [r3, 0]
    ldi  r5, 0xBE
    cmp  r1, r5
    bne  report
    addi r9, 0x01

report:
    ldi  r5, NUM_TESTS
    cmp  r9, r5
    bne  fail

pass:
    ldi  r1, MAGIC
    ldi  r3, RESULT_ADDR
    st   r1, [r3, 0]
    halt

fail:
    ldi  r1, 0x00
    ldi  r3, RESULT_ADDR
    st   r1, [r3, 0]
    halt
```

math.asm — `multiply` (shift-and-add)

```
.import mul_ret
.export multiply

multiply:
    push r5
    ldi  r3, 0x00              ; accumulator
    ldi  r4, 0x01              ; bit mask
mul_loop:
    ldi  r5, 0x00
    cmp  r2, r5
    beq  mul_done
    mov  r5, r2
    and  r5, r4
    ldi  r6, 0x00
    cmp  r5, r6
    beq  mul_skip_add
    add  r3, r1
mul_skip_add:
    mov  r5, r1
    add  r1, r5                ; shift left
    ldi  r5, 0x01
    shr  r2, r5                ; shift right
    br   mul_loop
mul_done:
    mov  r1, r3
    pop  r5
    br   mul_ret               ; ← RELOCATION
```

io.asm — peek / poke

```
.import io_ret
.export emit_byte
```

```
.export read_byte

OUT_PORT: .imm 0x01
IN_PORT:  .imm 0x02

emit_byte:
    poke r1, OUT_PORT
    br   io_ret                ; ← RELOCATION

read_byte:
    peek r1, IN_PORT
    br   io_ret                ; ← RELOCATION
```

Assembly Results

After running `atlas asm` on each file:

Object file	`.text` size	Symbols	Relocations
`main.o`	134 bytes (67 instr.)	26	1 (`multiply`)
`math.o`	96 bytes (48 instr.)	13	4 (`mul_ret`, `div_ret`, `abs_ret` ×2)
`io.o`	8 bytes (4 instr.)	5	2 (`io_ret` ×2)

Observations

- **main.o** has only 1 relocation despite importing 5 symbols — the other 4 are declared but never directly referenced as branch targets.
- Constants like `STACK_TOP` and `RESULT_ADDR` produce **zero relocations** — they are resolved to numeric values during assembly.

Linking — Section Merging

The linker concatenates `.text` sections in input order:

```
Merged .text (238 bytes):
┌──────────────────────────┬────────────────────────┬──────────────┐
│ main.o .text (134 bytes) │ math.o .text (96 bytes) │ io.o .text   │
│ base = 0x0000            │ base = 0x0086           │ base=0x00E6  │
└──────────────────────────┴────────────────────────┴──────────────┘
```

Linking — Symbol Resolution

Selected symbols after base adjustment:

Symbol	Source	Local offset	Section base	Final address
`main`	`main.o`	`0x0000`	`0x0000`	**`0x0000`**
`mul_ret`	`main.o`	`0x004A`	`0x0000`	**`0x004A`**
`io_ret`	`main.o`	`0x0082`	`0x0000`	**`0x0082`**
`multiply`	`math.o`	`0x0000`	`0x0086`	**`0x0086`**
`divide`	`math.o`	`0x0028`	`0x0086`	**`0x00AE`**
`abs_value`	`math.o`	`0x004C`	`0x0086`	**`0x00D2`**
`emit_byte`	`io.o`	`0x0000`	`0x00E6`	**`0x00E6`**
`read_byte`	`io.o`	`0x0004`	`0x00E6`	**`0x00EA`**
`RESULT_ADDR`	`main.o`	—	—	**`0x0080`** (abs)
`STACK_TOP`	`main.o`	—	—	**`0x00F0`** (abs)

Linking — Relocation Patching

Here’s how each of the 7 relocations gets patched:

#	File	Offset	Symbol	Patch offset	Resolved	Before → After
1	`main.o`	`0x0048`	`multiply`	`0x0048`	`0x86`	`0x8800` → `0x8886`
2	`math.o`	`0x0022`	`mul_ret`	`0x00A8`	`0x4A`	`0x8800` → `0x884A`
3	`math.o`	`0x003E`	`div_ret`	…	`0x7A`	`0x8800` → `0x887A`
4	`math.o`	`0x0046`	`div_ret`	…	`0x7A`	`0x8800` → `0x887A`
5	`math.o`	`0x005A`	`abs_ret`	…	`0x7E`	`0x8800` → `0x887E`
6	`io.o`	`0x0002`	`io_ret`	`0x00E8`	`0x82`	`0x8800` → `0x8882`
7	`io.o`	`0x0006`	`io_ret`	`0x00EC`	`0x82`	`0x8800` → `0x8882`

Pattern

In every case the linker keeps the upper byte (`0x88` = unconditional branch opcode) and writes the resolved 8-bit address into the lower byte.

Final Output

```
$ atlas asm test/main.asm test/main.o
  Assembled test/main.asm → test/main.o
    (134 bytes, 26 symbols, 1 relocations)

$ atlas asm test/math.asm test/math.o
  Assembled test/math.asm → test/math.o
```

```
    (96 bytes, 13 symbols, 4 relocations)

$ atlas asm test/io.asm test/io.o
  Assembled test/io.asm → test/io.o
    (8 bytes, 5 symbols, 2 relocations)

$ atlas ld test/main.o test/math.o test/io.o -o test/output.hex
     Linked 3 objects → test/output.hex (238 bytes)
```

The 238-byte linked image contains a fully self-testing program: it runs 6 tests covering addition, subtraction, bitwise logic, shifts, cross-module multiplication, and memory load/store, then writes a sentinel value to memory address `0x80` indicating pass (`0xAA`) or fail (`0x00`).