Beethoven CLI – Cheatsheet

The companion command-line tool for managing, building, and running Beethoven projects – from a one-line scaffold to simulation, FPGA synthesis, and AWS F2 deployment.
Usage: `beethoven [OPTIONS] <COMMAND>` • `https://github.com/Composer-Team/Beethoven-Software`

The Beethoven repositories

Zoo … ready-to-build example projects, one per folder
Software .. the toolchain: `libbeethoven` (C++ API), `cli` (Rust), `runtime` (daemon)
Hardware the Chisel framework every project depends on for RTL + C++ bindings

Pre-compiled CLI binaries (Linux x86-64/arm64, macOS arm64):
`github.com/Composer-Team/Beethoven-Software/releases`

Environment setup

You need a few system packages, a Java/Scala/sbt toolchain, and a one-time `beethoven setup`.

1. System packages.

Ubuntu / Debian:

```
sudo apt-get update
sudo apt-get install -y \
  curl build-essential vim zip unzip \
  git cmake iverilog verilator rsync \
  openssh-client
```

RHEL / Rocky / Fedora (iverilog & verilator live in EPEL):

```
sudo dnf install -y epel-release
sudo dnf groupinstall -y "Development Tools"
sudo dnf install -y \
  curl vim zip unzip git cmake \
  iverilog verilator rsync openssh-clients
```

macOS (Homebrew):

```
brew install icarus-verilog verilator \
  cmake git rsync
xcode-select --install   # build tools
```

2. Java, Scala & sbt (via SDKMAN).

```
curl -s "https://get.sdkman.io" | bash
source "$HOME/.sdkman/bin/sdkman-init.sh"
sdk install java 17.0.9-graalce
sdk install scala
sdk install sbt 1.10.11
sdk default sbt 1.10.11
```

3. Bootstrap Beethoven (clones Beethoven-Software/Hardware, installs `libbeethoven`; run once from any directory):

```
$ beethoven setup
```

Quick start

```
beethoven new my-project   # pure-chisel
#  ...or:  beethoven new --verilog my-project
cd my-project
beethoven build            # hw + runtime + sw
beethoven sim              # run testbench in sim
```

`-verilog` scaffolds a project whose hardware top is a hand-written `<Accel>Core.v` blackbox; the framework auto-syncs its port list against the Scala-side command struct on every build.

Project structure

```
my-project/
  Beethoven.toml   # manifest (you edit this)
  build.sbt        # generated -- don't edit
  hw/              # Chisel/Scala sources
  sw/              # C++ testbench + CMakeLists
  target/          # all build outputs (gen'd)
    binding/       # generated C/C++ bindings
    simulation/    # sim RTL + runtime + sw
    synthesis/     # synth outputs (if capable)
```

`hw/` ..... your accelerator; override with `[hardware] src-dir`
`sw/` ..... C++ host/testbench; override `[software] src-dir`
`Beethoven.toml` . manifest; CLI generates sbt + cmake from it

What `target/` holds:

`binding/` ..... `beethoven_hardware.h/.cc`: reg addrs, CSR enums, AXI-Lite offsets (mode-agnostic)
`simulation/` ..... Chisel lowered to SV, sim daemon, compiled tb
`synthesis/` ..... synth-optimized RTL + Xilinx/AWS collateral + hw daemon (`-release`)

Project config: Beethoven.toml

Analogous to `Cargo.toml`; commented keys show defaults, uncomment to override.

```
[project]
name = "my-project"           # required

[hardware]
# src-dir = "hw"              # default "hw"

[hardware.beethoven-hardware] # framework dep
version = "latest.integration"  # newest
# version = "0.1.7-dev12"     # pin a release
# path = "../Beethoven-Hardware"  # source-link

# [software]
# src-dir = "sw"
# default-testbench = "..."   # if many exist

[platform]
target = "default"            # required
```

Platform targets (Synth? = supports synthesis):
`default` ..... no synth; fast sim harness, usual choice
`kria` ..... synth; Xilinx Kria SoC (real silicon)
`aupzu3` synth; AUP-ZU3 – needs `dram-size-gb = 4|8`
`aws-f1` ..... synth; needs `memory-channels`
`aws-f2` ..... synth; remote SSH synth flow
`u200/u250/u280` ..... synth; Xilinx Alveo cards

Simulator *is set per platform table: `simulator = "icarus"` (default) or `"verilator"` (faster). Build* mode *is not in the manifest – it's chosen by the command (`sim` vs `run`, `build` vs `build -release`).*

Build: beethoven build [hw|runtime|sw]

Builds hardware, runtime daemon, and user sw. Pass a slice to build one part; omit to build all three.

(none) .. synth-capable targets build both sim & synth
`-release` ..... synthesis only (`target/synthesis/`)
`-simulation` ..... simulation only (conflicts `-release`)
`-j, -jobs <N>` ..... parallel jobs (default: all cores)

```
beethoven build              # everything
beethoven build hw --release # synth RTL only
beethoven build sw           # recompile tb
```

Companions: ***`beethoven check`*** *(validate toml + elaborate, no codegen);* ***`beethoven info`*** *(resolved config,* ***`-format text|json`****);* ***`beethoven clean`*** *(wipe* ***`target/`****).*

Simulation: beethoven sim [tb]

Builds and runs end-to-end in sim, auto-detecting a running daemon. Omitted `tb` uses the default/only testbench.

`-no-build` ..... run what's already built
`-no-launch` ..... require a daemon already up
`-simulator <s>` .. override backend (icarus | verilator)
`- <args>` forwarded to the testbench (like `cargo run`)

```
beethoven sim
beethoven sim my_tb -- --iterations 100
```

Runtime management

The daemon is started/stopped automatically by `sim` / `run`. Manage it by hand only when debugging:
`beethoven runtime <sub>`
`build` ..... cmake-build the daemon (`-release`, `-j N`)
`run` .... foreground daemon; Ctrl-C to stop (`-release`, `-log-file`)
`kill` ... SIGKILL daemon; wipe lockfile, shmem, build dirs (idempotent)
`clean` .... rm `target/<mode>/runtime/` (`-release` for synth)

```
beethoven runtime run --log-file rt.log
# ...in another shell:
beethoven sim --no-launch   # attach to it
beethoven runtime kill      # tear it down
```

Real-FPGA run: beethoven run [tb]

Like `sim` but executes on a real FPGA. Flags mirror `sim` (`-no-build`, `-no-launch`, `- <args>`).

Real-FPGA runs need ***`/dev/mem`*** *– run under sudo, preserving env:*

```
sudo -E env HOME=$HOME beethoven run -- --foo bar
```

ZYNQ flow: synth & flash

`beethoven build -release` emits Vivado TCL under `target/synthesis/implementation`. Source Vivado, then run the full setup/synth/impl/bitstream pipeline:

```
source /opt/Xilinx/2025.2/Vivado/settings64.sh
beethoven synth
```

`-no-build` ..... skip prereq `build hw -release`
`-up-to <st>` ..... stop after `setup|synth|impl|bit` (default `bit`)
`-gui` ..... open Vivado GUI on `0_setup.tcl`, then exit

`synth` *always does a full rebuild –* ***`0_setup.tcl`*** *wipes* ***`xilinx_work/`*** *first.*

Flashing. `beethoven flash` JTAG-programs the FPGA with the bitstream from `synth`.

```
$ beethoven flash
```

Needs a Xilinx ***`hw_server`*** *reachable at* ***`localhost:3121`****, run on the machine the JTAG cable is attached to.*

AWS F2 helpers: beethoven aws <sub>

upload – rsync the CL package (`target/synthesis/aws/cl_beethoven_top/`) to a remote builder. Uploads only; does *not* start the build.
`-host <user@ip>` ..... SSH destination (required)
`-key <path>` ..... SSH private key
`-remote-dir <d>` ..... default `~/cl_beethoven_top`
`-delete` ..... delete remote files absent locally

create-fpga-image – run on the F2 builder after Vivado finishes; validates timing, uploads checkpoint + design-env tar to S3, calls `aws ec2 create-fpga-image`.
`-name <name>` ..... AFI name / S3 key stem
`-region <r>` ..... else from `AWS_REGION` / CLI config
`-bucket <s3>` ..... S3 bucket for AFI input + logs
`-auto` ..... pick newest valid artifact
`-yes` ..... non-interactive; fail vs prompt
`-dry-run` ..... print plan, don't run

load – load a created image into a local F2 slot. Does *not* build or start the runtime.
`-agfi <agfi>` ..... specific AGFI (vs `-afi/-name`)
`-afi <afi>` ..... specific AFI
`-name <name>` ..... select available AFI by name
`-slot <N>` ..... FPGA slot (default `0`)
`-region <r>` ..... AWS region

End-to-end AWS F2 workflow

Three machines: your **local** box, a **compile VM** (synthesis only), and the **F2 instance** (the real FPGA).

Wherever credentials appear, use your own AWS key / secret / region – never commit them.

A. Configure AWS credentials (local; AWS CLI installed first):

```
aws configure   # key, secret, region, json
# install if needed (x86_64):
curl "https://awscli.amazonaws.com/\
awscli-exe-linux-x86_64.zip" -o awscliv2.zip
# arm64 -> awscli-exe-linux-aarch64.zip
unzip awscliv2.zip && sudo ./aws/install
```

B. Local: build & upload the CL package:

```
# Beethoven.toml: [platform] target="aws-f2"
beethoven build hw --release
beethoven aws upload \
  --host ubuntu@$AWS_VM_PUBLIC_IP \
  --key /path/to/key.pem
```

C. On the compile VM: synthesize & create the AFI:

```
# copy creds up so create-fpga-image works:
scp -i /path/to/key.pem \
  ~/.aws/credentials ~/.aws/config \
  ubuntu@$AWS_VM_PUBLIC_IP:~/.aws/
ssh -i /path/to/key.pem ubuntu@$AWS_VM_PUBLIC_IP
# --- now on the VM ---
sudo apt-get update && sudo apt-get install -y \
  cmake build-essential git curl tmux python3
beethoven setup --no-hardware
tmux new -s f2-build      # synth ~1.5 hr
cd ~/cl_beethoven_top/
source /opt/Xilinx/2025.2/Vivado/settings64.sh
./build_beethoven_f2.sh
beethoven aws create-fpga-image \
  --name my-image --region us-east-1 --yes --auto
```

Record the **AGFI** it prints. Confirm the image is available before using it:

```
aws ec2 describe-fpga-images --owners self \
  --filters Name=name,Values=my-image \
  --query 'FpgaImages[0].{State:State.Code,\
AGFI:FpgaImageGlobalId}' --output json
```

No teardown needed – the compile VM's lifecycle is managed for you. Once the AGFI is available you're done with it.

D. On the F2 instance: load & run:

```
# project uploaded as ~/my-project
beethoven setup --no-hardware
cd ~/my-project
sudo beethoven aws load \
  --agfi agfi-XXXXXXXXXXX --slot 0  # your agfi
sudo beethoven run
```