B5 — B-Stage Tape-Out Preparation & Assessment

Congratulations! You’ ve basically met the B-stage tape-out targets. Next, you still need to complete some tape-out related preparation work.

Pre–tape-out preparation

Switch to 32-bit ysyxSoC

The SoC team has provided a 32-bit tape-out SoC. We have also modified ysyxSoC to 32-bit to help everyone test locally before integrating with the tape-out SoC. Compared with the previous 64-bit ysyxSoC, the 32-bit version mainly changes the bus data width to 32 bits and removes the AXI data-width conversion module.

Switching to 32-bit ysyxSoC

If you cloned the ysyxSoC project after **2024/07/26 13:00:00**, you don’ t need to do anything. Otherwise, do the following:

- Run the command below to get the new 32-bit ysyxSoC:
- `cd ysyxSoC`
- `git pull origin master` (you may need to resolve code conflicts)
- Change the AXI data width at the NPC top level to **32 bits**, and remove the related data-width conversion code.
- Re-run simulation using the 32-bit ysyxSoC environment.

Open NPC address space

The tape-out SoC may integrate more devices than ysyxSoC. If NPC blocks accesses to certain address ranges in advance, it won’ t be able to access those devices after tape-out. Therefore, we recommend opening **all** address space in NPC so that all read/write requests are sent out through NPC’ s external AXI bus interface. If an address is accessed where no device exists, the SoC will return an error through AXI’ s `resp` signal.

Remove negative-edge-triggered clocks

Mixing positive-edge and negative-edge clocking makes timing closure harder and increases back-end physical implementation difficulty. If your processor severely impacts the overall SoC timing, and the schedule is tight, the “One Student One Chip (一生一芯)” project team will remove your processor from the tape-out list for that batch.

Remove negative-edge-triggered clocks

Check whether your code contains any negative-edge-triggered clocks. If so, modify the corresponding modules to remove them.

Remove latches

In synchronous sequential logic circuits, all storage element access is controlled by the clock. But latch writes are not clock-driven, making them hard for timing analysis tools to analyze, so they are generally not used in synchronous sequential circuits.

Remove latches

Use **yosys** to synthesize your design, then check whether `synth_stat.txt` contains any of these standard cells:
`DLL_X1`, `DLL_X2`, `DLH_X1`, `DLH_X2`.
If yes, it means your design contains latches, and you must modify the corresponding modules to remove them.

Chisel perk: Verilog generated by Chisel describes synchronous sequential logic, so if you develop in Chisel, you don’t need to worry about generating latches.

Naming changes

Multiple students’ code will be integrated into one SoC. If module names clash, EDA tools will report “duplicate module definition” errors. To avoid this, make these changes:

- Merge your CPU code into a single `.v` file named `ysyx_<8-digit student ID>.v`, e.g. `ysyx_22040228.v`.
- On Linux you can do this with `cat`:
- `cat CPU.v ALU.v regs.v ... > ysyx_22040228.v`
- Rename the CPU top module to `ysyx_<8-digit student ID>`, e.g. `ysyx_22040228`.
- Add the prefix `ysyx_<8-digit student ID>_` to all variables defined using `define` inside the CPU.
- Example: change `define SIZE 5` to `define ysyx_22040228_SIZE 5`

- If you use Chisel, you don’t need this part.
- Add the prefix `ysyx_<8-digit student ID>_` to all module names inside the CPU.
- Example: `module ALU` → `module ysyx_22040228_ALU`

Chisel perk (2): If you use Chisel, you can add an Annotation to automatically add module-name prefixes (see the discussion in the referenced issue). If you use Verilog/SystemVerilog, automatic prefixing isn’t supported for now—please add it manually.

Static code checking

Verilator can perform static linting on Verilog and warn about potential risks. Fixing these helps improve correctness. Use Verilator’s `--lint-only` option.

To enable more checks, add `-Wall` and `-Werror`. Verilator’s manual documents all warnings. In principle, you should clear as many as possible to make your code more standard—except for these two warning categories:

- `DECLFILENAME` warnings are unrelated to logic; ignore them via `-Wno-DECLFILENAME`.
- `UNUSED` warnings:
- Some are because input ports are unused (e.g., top-level `io_slave_arvalid`) and can be ignored.
- Others may indicate messy code. You must carefully confirm the cause. If you choose to ignore a warning, you must understand the risk and take responsibility.

Chisel perk (3): Chisel-generated Verilog is usually规 范 /standard-compliant, but may still produce `UNUSED` warnings related to signals and bit widths. Chisel may make this step easier, but you should still check every warning carefully.

4-value simulation (X-propagation checking)

System startup includes cold boot (power-off → power-on + reset) and warm reset (already powered → reset). Circuit state depends on sequential elements, so after reset the circuit should enter a correct expected state. This requires specifying reset values for sequential elements.

One approach is to reset all sequential elements so cold boot and warm reset end in the same state. But this is impossible for circuits with RAM: for example DRAM storage arrays have no reset; even SRAM arrays often have no reset. This means in warm reset, RAM contents persist across reset. Therefore, the circuit must work correctly regardless of what data RAM holds after reset. In other words, reset-time behavior must be independent of RAM contents.

Besides RAM, remaining sequential elements are flip-flops. Resetting all flip-flops increases probability of correct state, but adds delay and area:

- Reset signal routing increases reset propagation delay.
- Reset-capable flip-flops require extra gates/area.

In real projects, engineers reset only a minimal subset of flip-flops that is sufficient for correct post-reset behavior. Flip-flops that don’t affect post-reset behavior do not need reset.

Typically, flip-flops that don’t need reset fall into two categories:

1. **Registers that software can initialize before use** (software-visible ISA-defined regs).
Example: in RISC-V, general-purpose registers (except x0) are unspecified after reset, so software should write them before reading. Similarly `mepc` is unspecified after reset; software should only read it after it has been written by an exception or by CSR writes. Setting them as “undefined after reset” moves initialization burden from hardware to software, saving area and delay.

2. **Data-path flip-flops guarded by control signals** (like `valid`, `ready`, `en`).
If control indicates data is invalid, downstream logic won’t use/store the data. So most data-path flip-flops can be left unreset, as long as when you assert the associated control, those flip-flops already contain valid data. Example: pipeline registers holding load data may not need reset if the corresponding `valid` register is reset.

Resetting flip-flops that don’t need reset mostly wastes area and increases reset delay. But failing to reset flip-flops that do need reset can cause incorrect behavior after reset—this must be avoided.

Previously you used Verilator, which is a **2-value** simulator (signals are only 0/1). So every flip-flop effectively has a concrete value after reset, which doesn’t reveal the issue well.

In **4-value** RTL simulators, an unknown initial value of a flip-flop without reset is represented as `X`. `X` can propagate through logic. If a flip-flop that should have been reset is not reset, `X` may propagate widely, causing many signals to become `X`, and the circuit may converge to an unintended state—programs will not run as expected. Conversely, if all necessary flip-flops are reset, `X` propagation is blocked by control signals and shrinks as valid data updates, eventually converging to the expected correct state.

Using iverilog for 4-value simulation

`iverilog` supports 4-value simulation. Install it first:

- `apt-get install iverilog`

Goal: check whether NPC can run programs under iverilog. Since ysyxSoC is not taped out, you only need to simulate NPC alone and run the AM program for `riscv32e-npc`.

However, iverilog currently doesn't support DPI-C or driving simulation via C++ code, so you need changes to the simulation environment:

- Remove DiffTest and memory-related DPI-C calls.
- Make NPC execute from `0x80000000`. Possible methods include (not limited to):
- Set PC reset value to `0x80000000`, or
- Keep PC reset value as `0x30000000`, but place instructions at `0x30000000` that immediately jump to `0x80000000`.
- Re-implement memory read/write in Verilog or Chisel. Since this memory is only for simulation, it doesn't need to be synthesizable; behavioral modeling is fine.
- Initialize memory via `$readmemh()` by loading a memory image file.
- `$readmemh()` input format can be generated via `objcopy -O verilog`.
- But `$readmemh()` assumes memory addresses start at 0, unlike ELF link addresses, so you also need `--adjust-vma` (see `man objcopy`).
- Write a simple simulation top module that instantiates clock/reset and drives NPC.
- When compiling with iverilog, the macro `__ICARUS__` is automatically defined, so you can wrap these changes in conditional compilation so your code supports both verilator and iverilog.

After the changes, try compiling with iverilog. Since iverilog supports less syntax than Verilator, use `-g2012` for a newer standard. You may still encounter cases where Verilator compiles but iverilog doesn't; fix code according to errors.

If you use Chisel, add firtool options:

- `--lowering-options=disallowLocalVariables,disallowPackedArrays`

to avoid language features unsupported by iverilog.

You may also see the following warning, which can be ignored:

- `sorry: constant selects in always_* processes are not currently supported (all bits will be included).`

Since DiffTest can't be used, if results are unexpected you must diagnose via waveforms. Therefore, we recommend eliminating as many issues as possible using Verilator + DiffTest first, then using iverilog. See iverilog docs for waveform generation.

Check X-propagation via 4-value simulation

Simulate NPC in iverilog and run programs like microbench and RT-Thread. If program execution fails, modify the RTL to fix X-propagation issues.

Netlist simulation

Verilog is event-driven, and synthesizable Verilog is only a subset. Simulators may accept non-synthesizable code; that’ s not a simulator bug—it's standard-defined behavior. In RTL design, we don’ t want to write non-synthesizable code.

To detect non-synthesizable code, simulate the **post-synthesis** circuit. If non-synthesizable constructs exist, the synthesized circuit may differ from RTL, revealing errors. So you can simulate the Yosys synthesis result.

Synthesis converts RTL into logically equivalent standard cells. The file describing their connections is a **netlist**. To simulate it, you need module-level models for those standard cells. The yosys-sta project provides them at:

- `yosys-sta/nangate45/sim/cells.v`

Netlist simulation

Simulate using the synthesized netlist plus the standard-cell simulation models, and try running microbench, RT-Thread, etc. Since memory outside the AXI interface is not part of tape-out scope, you can keep using the memory code from 4-value simulation. It’ s recommended to keep using iverilog.

Chisel perk (4): Chisel-generated Verilog is synthesizable, so netlist simulation will likely succeed directly. Still, don’ t skip netlist simulation.

Re-check your code

If you modified code during the above steps, repeat the checks to avoid re-introducing non-compliant code.

Apply for the code debugging assessment

Process may still change

The application process is currently in internal testing and may be updated—please stay tuned.

You can scan the QR code below to add the TA on WeChat and join the internal test group.

Before applying, submit your code in the required way. We will use a CI pipeline to run checks. When your code passes all checks, CI will automatically apply for the code debugging assessment for you.

Because CI is automated:

- If a check fails, read the logs, fix the issue, and resubmit.
- If CI fails due to simulation results not matching expectations, CI won’t provide the failure scene or waveforms. So you should do thorough local testing and debugging before submitting.

Since CI is automated, it assumes certain directory structure, file naming, and make targets. You must adjust your project accordingly. These adjustments might affect your existing simulation workflow; if the impact is too large, you can create a new branch in ysyx-workbench, apply adjustments there, and submit that branch only.

After the adjustments, push the new branch to a public repository. Then provide the repository URL and branch name in the specified way so CI can pull and test your code.

1) Tool version requirement

CI uses Verilator from the **stable** branch. Since stable evolves, the CI version may differ from your local version. Usually it’s fine, but compatibility issues can occur (e.g., code compiles locally but not in newer stable). If so, adapt your code to the latest stable.

2) Directory structure requirement

Your ysyx-workbench must contain:

```
ysyx-workbench
├── abstract-machine
├── nemu
├── npc
├── Makefile
└── patch
    ├── rt-thread-am
    │   ├── 0001.aaa.patch
    │   ├── 0002.bbb.patch
    │   └── ...
    └── ysyxSoC
        ├── 0001.xxx.patch
        ├── 0002.yyy.patch
        └── ...
```

Notes:

- The top folder name can be different; CI will rename it to `ysyx-workbench` after cloning. So the remote repo name doesn't matter.
- `abstract-machine`, `nemu`, and `npc` must not be renamed.
- `Makefile` must not be renamed; `STUID` and `STUNAME` must be set correctly.
- For `rt-thread-am` and `ysyxSoC`, CI uses patches to apply your changes:
- CI re-clones upstream repos and applies patches under `patch/`.
- See "How to generate patches" below.
- CI testing does not depend on your modifications to `am-kernels`, `nvboard`, or `yosys-sta`, so CI will re-clone upstream versions and will not include your changes to those projects.

3) Make target requirements

CI expects the following:

- `make -C npc verilog` generates `ysyx_xxxxxxxx.v` or `ysyx_xxxxxxxx.sv` under `npc/build/` for SoC integration.
- PC reset value is `0x30000000`

- xxxxxxxx is your 8-digit “One Student One Chip” ID
- File must contain a module named ysyx_xxxxxxxx
- The character before module ysyx_xxxxxxxx must be whitespace (starting it on a new line is acceptable)
- The CLINT module address matches CLINT in ysyxSoC
- If both .v and .sv exist, CI prefers .sv
- CI runs make -C ysyxSoC dev-init then make -C ysyxSoC verilog to generate ysyxSoCFull.v.
- After make -C ysyxSoC verilog, CI assumes ysyxSoC/build/ysyxSoCFull.v has ysyx_00000000 modified so 00000000 becomes your student ID.
- In an am-kernels code directory:
- make ARCH=riscv32e-npc builds a program for NPC without SoC; first instruction at 0x80000000.
- make ARCH=riscv32e-ysyxsoc run builds a program for NPC with SoC integrated and starts simulation; first instruction at 0x30000000.
- make -C npc sim-iverilog IMG=xxx uses iverilog to simulate the above ysyx_xxxxxxxx.v/.sv and run program xxx.
- Program first instruction at 0x80000000
- Note: xxx is a .bin file, but readmemh() can’t read .bin directly. You can add commands in this rule to convert .bin into a readmemh()-compatible file before sim.
- make -C npc sim-iverilog-netlist IMG=xxx NETLIST=yyy CELLS=zzz uses iverilog to simulate netlist yyy and run program xxx (0x80000000).
- yyy: netlist produced by yosys-sta synthesis of ysyx_xxxxxxxx.v/.sv
- zzz: path to standard cell behavioral simulation file; added to the simulation source list
- xxx is also a .bin file

Additionally:

- CI will not test DiffTest; we recommend disabling DiffTest.
- If you want DiffTest enabled, you must insert commands to compile NEMU when building.
- Beyond the framework’s provided make targets, CI won’t run other make targets.
- Also disable waveform dumping; CI won’t return waveforms. Waveforms slow simulation and don’t help CI. CI is not a debugging platform—debug locally.

4) How to generate patches

As noted, CI uses patches for your `rt-thread-am` and `ysyxSoC` modifications. You must generate patches manually using `git format-patch`.

Example:

- `cd rt-thread-am`
- `git format-patch origin/master`

This generates patch files like `0001-xxx.patch`, one per commit, covering all commits from upstream `master` to your local branch `HEAD`.

Move those patch files to:

- `ysyx-workbench/patch/rt-thread-am/`

CI will start from upstream `master` and apply patches sequentially to reproduce your local state.

Notes about patches

- Do not manually modify intermediate build artifacts (compiled outputs, etc.), otherwise CI will overwrite them. If you need to change generated results, add the change as a command in the build process (e.g., Makefile).
- Total patch size must not exceed **1MB**. If it does, you probably tracked unnecessary files (like build outputs) in git, indicating improper git usage. In that case, rebuild a clean branch history (STFW).

5) Repository structure requirement

After preparing ysyx-workbench, push it to a **public** repo. Branch name can be anything. In addition, you must push the `tracer-ysyx` branch to the same repo. So the repo must contain at least two branches:

- A branch (any name) that meets the directory structure requirements
- `tracer-ysyx` (auto-maintained by the tracing system)

Any hosting platform is fine as long as both branches can be cloned via `git clone` over HTTPS.

6) Preparation for the code debugging assessment

After CI passes, the TA will directly pull the CI-tested code as the basis for the debugging assessment. So you must ensure NEMU can successfully run RT-Thread. Also avoid deleting code that helps you debug. If that code affects area evaluation, guard it with macros: disable it during CI, enable it manually during the debugging assessment.

7) Add your student ID to the whitelist

After finishing the above, contact the TA to add your student ID to the CI whitelist.

8) Fill out the assessment application form

Next, fill out the application form. Visit the repo:

`https://github.com/未发布`

Create a new issue, select the assessment template, and fill in the fields as guided:

- **Title**: For easier searching, include your 8-digit “One Student One Chip” student ID.
- **Student ID**: 8 digits, without `ysyx_` etc.
- **Repo URL**: link to your repo described above
- **Branch name**: the branch that meets the directory structure requirements
- **Comment**: any marker for this submission, e.g. “5th submission”, “fixed directory”, etc. CI will include this comment in its returned results to help you distinguish multiple submissions.

After you submit the issue, CI is triggered and will comment under the issue with a timestamp, your comment, and the workflow URL. You can click the workflow URL to view CI progress.

- If CI succeeds: it appends a line “Finish” and closes the issue. Your code passed CI and you can proceed to the next steps of the debugging assessment.
- If CI fails: it appends “End with errors”. Click the workflow URL, find the failing job/step, fix and push your code, then edit the issue to update the comment and resubmit to trigger CI again.

If CI succeeds, it will push your code to their repo under a branch named `ysyx_<studentID>`. If that branch already exists, the push fails. This means the current process does not allow overwriting code that has already passed CI—only **one successful CI run** is allowed. If you need to update code after passing CI, contact the TA and explain why.

CI execution takes more than half an hour. When many students submit, the more-tests job may queue. Each CI job has a default max runtime of 6 hours; if it times out, it will be forcibly canceled. If a job is canceled due to long queue time, wait a while and then trigger a new CI run by editing the issue.

Participate in the code debugging assessment

After passing CI, you can take the final code debugging assessment. Please follow the assessment process instructions.