

This document is intended to provide general guidelines for what to do to add a design to our benchmark suite.

## Selecting a design

We want open-source designs that are representative of modern cores and accelerators. We don't want "yet another RISC-V core". We have a [spreadsheet](#) of some designs we think might be good.

We want the design to have an appropriate amount of verification infrastructure. While we aren't using this right now, we will likely add a verification component to our flow later. It is also indicative of a high-quality design to have a verification flow.

We want to have designs with different methodologies. Many of them will be Verilog/SystemVerilog, but others might be generated from Chisel, Python, etc.

## Setting up the suite

Follow the instructions on the [suite's github](#) (which is wrapped around OpenROAD flow scripts), to get set up.

## Target technology

Our target technology is, first and foremost, ASAP7. Secondly, we want NANGATE45 and finally SKY130HD.

## Suite features

### Understanding the suite's **Makefile**

Viewing the [Makefile](#) helps to make sense of how we intend to use the suite to port designs. It's important to understand the following snippet:

```
do-dev-setup:  
    git submodule init $(BENCH_DESIGN_HOME)/src/$(DEV_DESIGN_HOME)/repo  
    git submodule update $(BENCH_DESIGN_HOME)/src/$(DEV_DESIGN_HOME)/repo  
    # Check if a setup.sh script exists for the current design  
    @if [ -f "$(BENCH_DESIGN_HOME)/src/$(DEV_DESIGN_HOME)/setup.sh" ]; then \  
        bash $(BENCH_DESIGN_HOME)/src/$(DEV_DESIGN_HOME)/setup.sh; \  
    fi
```

This `do-dev-setup` is intended for those who are actively porting designs (see [here](#) for dev vs user info), and assumes that the developer has already added the design repository as a git submodule to the repo's `.gitmodules` file.

Assuming that we've added the specified design's repository as a git submodule (matching the correct path `$(BENCH DESIGN HOME)/src/$(DEV DESIGN HOME)/repo`), this `do-dev-setup` will initialize and update the repo. Then, it'll check if a `setup.sh` file exists in the dev folder of the design. This `setup.sh` file is intended to convert the design's RTL files to Verilog, preparing all the RTL files for running in the OpenROAD flow (that is if it's in another language, otherwise the `setup.sh` file may not be needed).

## dev vs user suite usage

Before we move on to the process of porting a design, we should note that this suite is intended to be used by both developers (those who are actively adding designs to the suite) and users (those who use our designs for evaluation, training, etc.) in tandem. However, there are a few differences in navigating the suite as a developer compared to as a user.

### Suite access as a developer

As a developer, we're responsible for preparing **Verilog** RTL files (which may be converted from another language like SystemVerilog, Python, Chisel, etc.) that can be read by Yosys to synthesize and generate a floorplan for in ORFS.

The `setup.sh` script that we mentioned before is something that we as developers need to create, in order to convert a design (to **Verilog**) and apply any patches that may be needed for synthesis. This script will automatically get called in the Makefile environment when using `make dev`. Check out a `setup.sh` [example from minimax](#), one of the designs in our suite. It sets up sv2v, which is a software that lexically converts SystemVerilog files into Verilog.

We haven't yet generated the Verilog files from SystemVerilog, we'll handle that in our design's `verilog.mk` file (see more about `verilog.mk` [here](#)).

The `setup.sh`, design repository, patch files, and anything else that helps to generate the design to be used for OpenROAD, should reside in the `designs/src/<design-name>/dev` directory. As an example, please take a look at the [minimax dev folder](#).

The idea is that by running `make dev`, we will generate the required Verilog files and further run the design in the OpenROAD flow. Once generated, its Verilog files will be added to the `designs/src/<design-name>` directory so that users can access and generate the designs that we as developers have created.

## Suite access as a user

Assuming you're a user who wants to run a design created (and already generated) by a developer, you need only to run

```
make DESIGN_CONFIG=./designs/<technology-name>/<design-name>/config.mk
```

to begin the RTL-to-GDS flow for the specified design.

## designs/src/<design-name>/verilog.mk

The `verilog.mk` file is an addition to the suite which we use to reduce clutter (and redundancy) in the per-technology `config.mk` files. The `verilog.mk` file does a simple check to see if the suite was run using `make` or `make dev`, and responds accordingly.

You should add parameters to `verilog.mk` when they can be generalized to all technologies (like `ADDITIONAL_LIBS` and `ADDITIONAL_LEFS`), and `config.mk` when parameters are technology-specific (i.e. `CORE_UTIL`, `PLACE_DENSITY`, etc.).

Our minimax design has a [very simple example](#), which will run sv2v to generate Verilog files from SystemVerilog if `make dev` is called (which means `DEV_FLAG` is set). Otherwise, if `make` is called, we assume that the Verilog files have already been generated from SystemVerilog (i.e. the developer's job is finished), thus we only need to define `VERILOG_FILES` for ORFS.

## Porting a design to the suite: sha256 example

As a simple example and introduction to porting a design to the benchmark suite (i.e. the developer side of the suite), we'll use the hardware implementation of sha256, a cryptographic hashing function. It's a very small design in hardware and its runtime is very quick for ORFS, which is why we've opted for it as our example.

Before delving into per-technology requirements, we need to add the required RTL files from the [sha256 github repo](#) we're using (which reside in the `src/RTL` directory) to the `designs/src/<design-name>/dev` directory of the suite.

The minimum file requirements to run a design in **any technology** are:

- `config.mk` - defines the target platform, the source code for the design, and allows for modifying design flow parameters
- `constraint.sdc` - contains timing constraints for the design

We'll go over what each file does in the following sections.

## Making `designs/sha256/config.mk`

The config file requires that we define the following variables: `DESIGN_NAME`, `PLATFORM`, `VERILOG_FILES`, and `SDC_FILE`.

Since we're most interested in porting ASAP7 first, we'll be using it for the technology in the example.

It's up to us as designers to implement other variables to fit within desired parameters (i.e. timing, power, area). For example, you can modify a variable like `CORE_UTILIZATION` to allow the user to define how much of the design core area is used for the design, or `PLACE_DENSITY` to increase or decrease the density of cells during the placement stage. **A list of environment variables in which you can use to modify the design and varying stages of the flow can be found [here](#).**

`config.mk` for sha256:

```
export DESIGN_NAME = sha256    # Module name of top-level instance
export PLATFORM     = asap7      # Intended platform we wish to create design for

export VERILOG_FILES = $(sort $(wildcard ${DESIGN_HOME}/src/${DESIGN_NICKNAME}/*.v))
export SDC_FILE      = ${DESIGN_HOME}/${PLATFORM}/${DESIGN_NICKNAME}/constraint.sdc

export CORE_UTILIZATION = 40
export TNS_END_PERCENT = 100

export CTS_CLUSTER_SIZE = 25
export CTS_CLUSTER_DIAMETER = 45
```

\*NOTE: the design name, `sha256`, was **not** chosen at random. Looking at the [top level sha256 wrapper](#) `sha256.v`, we can see that it contains a module named `sha256` which instantiates the sha256 core. Also take note of the clock naming convention used in the design, which needs to be matched correctly in the designs `constraint.sdc` file.

## Making `designs/sha256/constraint.sdc`

The sdc file is where you'll define static timing information like clock speed and input/output delay. Information on creating SDC (Synopsys Design Constraint) files as well as varying STA constraints can be found [here](#).

Ensure that you use the correct clock name as described in the top level verilog file of the design.

`constraint.sdc` for sha256:

```
current_design sha256
```

```

current_design sha256

set clk_name clk
set clk_port_name clk
set clk_period 650
set clk_io_pct 0.25

set clk_port [get_ports $clk_port_name]

create_clock -name $clk_name -period $clk_period $clk_port

set non_clock_inputs [lsearch -inline -all -not -exact [all_inputs] $clk_port]

set_input_delay [expr $clk_period * $clk_io_pct] -clock $clk_name $non_clock_inputs
set_output_delay [expr $clk_period * $clk_io_pct] -clock $clk_name [all_outputs]

```

For ASAP7, `clk_period` is denoted by **picoseconds**. For NANGATE45 and SKY130, **nanoseconds** are used. Thus, the equivalent `clk_period` of 650 picoseconds in ASAP7 would be 6.5 nanoseconds for NANGATE45 and SKY130. Make sure to keep these per-technology differences in mind when porting.

This simple SDC file uses a percentage of the clock period for input and output delays which may not always make sense. If you have a very large or very small clock period, you may want to define these to a value based on the type of input or output signal you are constraining. **This is design dependent.**

## (Optional) designs/sha256/fastroute.tcl

By saying that this `fastroute.tcl` file is optional we mean that the design can make it to finish with or without it in the design directory. The reason we'd want to use `fastroute.tcl` is to modify the routing at some (or every) layer to lessen congestion. It also allows us to set specific layers for which we wish to route either signal or clock paths (we can still route both on the same layer).

Simple `fastroute.tcl` example:

```

set_global_routing_layer_adjustment $::env(MIN_ROUTING_LAYER)-$::env(MAX_ROUTING_LAYER) 0.4
set_routing_layers -clock $::env(MIN_CLK_ROUTING_LAYER)-$::env(MAX_ROUTING_LAYER)
set_routing_layers -signal $::env(MIN_ROUTING_LAYER)-$::env(MAX_ROUTING_LAYER)

```

## Running sha256 in the suite

Now that we have all the necessary files added to the suite for running a design, that is:

- `designs/src/sha256/dev/*.v` - all the required verilog files from the [sha256 secworks RTL](#)

- `designs/asap7/sha256/config.mk` - the `config.mk` file that we created above
- `designs/asap7/sha256/constraint.sdc` - the `constraint.sdc` that we created above

We should be able to run `make dev` which will attempt to call a design's `setup.sh` to generate verilog source files (if one exists), and then run the design from start to finish. If the design succeeds and makes it to finish, we make the RTL Verilog files accessible to users assuming that the design finishes with no issues (in the `designs/src/sha256/dev` directory).

## Generating memories (using FakeRAM)

One of the most common mistakes is to synthesize a memory or register file into DFFs. This will generally create a design that has suboptimal area, is quite congested, and may not even get past synthesis due to run-time and memory issues. Thus, on-chip memories are often specially designed, but it's a time-intensive and expensive process. It is commonly acceptable to synthesize memories if they are less than a hundred or so DFFs, but this is not a hard rule.

Instead, you should create a memory using our modified version of FakeRAM. The RAM designs we get from FakeRAM don't have any internal RAM logic, they simply represent a macro box with a reasonable area proportional to the size and geometries of pins for the RAM, which we can interface with the design we wish to port (this will be in a `.lef` file). FakeRAM also generates estimated timing measurements for the RAM designs (which belong to a `.lib` file). This means we can use FakeRAM to generate a design with accurate PPA metrics, without the hassle of creating specialized RAM for each design we port.

[Our FakeRAM repo](#) contains configs for ASAP7, NANGATE45, and SKY130. Follow the instructions on the [README](#) to get set up with custom FakeRAM configs.

## When to use FakeRAM for a design

It's very important that we know when to use FakeRAM for a design, as failure to do so can lead to a number of problems as discussed above. As an example, we'll show how we can generate FakeRAM for the [NyuziProcessor](#) design from our spreadsheet.

There's many different ways of searching through a design's source files for any SRAM modules that may need replacement with FakeRAM. As a starting point, we can try to search through the design's RTL files for the keyword "`sram`". This is the result:

```

hardware/core/sram_1r1w.sv
21  //
22  // Block SRAM with 1 read port and 1 write port.
23  // Reads and writes are performed synchronously. The read value appears
24
25  module sram_1r1w
26      #(parameter DATA_WIDTH = 32,
27
28      Show 4 more matches
29
30  hardware/core/sram_2r1w.sv
31
32  // Block SRAM with 2 read ports and 1 write port.
33  // Reads and writes are performed synchronously. The read value appears
34
35  module sram_2r1w
36      #(parameter DATA_WIDTH = 32,
37
38      Show 4 more matches
39
40  hardware/core/l2_cache_tag_stage.sv
41
42
43  sram_1r1w #(
44      .DATA_WIDTH($bits(l2_tag_t)),
45      .READ_DURING_WRITE("NEW_DATA")
46  ) sram_tags(
47      .read_en(l2a_request_valid),
48
49      Show 2 more matches

```

Already, we can see two modules that we'll need to replace with FakeRAM, `sram_1r1w` and `sram_2r1w`. We need to further search through the RTL files, replacing every instantiation of the sram with our FakeRAM. Using the keyword “`sram_1r1w`” to search again, we can see that one of the first instantiations of `sram_1r1w` occurs in [l2\\_cache\\_tag\\_stage.sv](#):

```

sram_1r1w #(
    .DATA_WIDTH(1),
    .SIZE(`L2_SETS),
    .READ_DURING_WRITE("NEW_DATA")
) sram_dirty_flags(

```

```
.read_en(l2a_request_valid),
.read_addr(l2a_request.address.set_idx),
.read_data(l2t_dirty[way_idx]),
.write_en(l2r_update_dirty_en[way_idx]),
.write_addr(l2r_update_dirty_set),
.write_data(l2r_update_dirty_value),
.*);
```

We're given the width of SRAM (which is 1 bit), so we just need to determine the value of `L2_SETS` to retrieve the depth. A quick lookup for `L2_SETS` leads us to the following snippet in the [config.svh](#) file:

```
`define L2_SETS 256          // 128k
```

The above instantiation of `sram_1r1w` is representative of a 1-bit width, 256-bit depth SRAM.

**Let's create a FakeRAM config so that it can be replaced:**

```
{
  "name": "fakeram_1x256_1r1w",
  "width": 1,
  "depth": 256,
  "banks": 1,
  "no_wmask": "true",
  "ports": {
    "r": 1,
    "w": 1,
    "rw": 0
  }
},
```

Once we've generated the above config, we need to place the resulting `fakeram_1x256_1r1w.lef` and `fakeram_1x256_1r1w.lib` files into the respective technology. The standard location in our suite for placing `*.lef` and `*.lib` macros for FakeRAM would be under the `designs/<technology-name>/<design-name>/sram/[lef, lib]` dirs. If there is already a memory of the appropriate size, you can reuse it.

To have the macros recognized by ORFS, we'll need to add the `lef` and `lib` file locations to the `ADDITIONAL_LEFS` and `ADDITIONAL_LIBS` parameters, under our designs `verilog.mk` file:

```
export ADDITIONAL_LEFS = \
$(BENCH DESIGN_HOME)/$(PLATFORM)/$(DESIGN_NAME)/sram/lef/fakeram_1x256_1r1w.lef
```

```
export ADDITIONAL_LIBS = \
$(BENCH DESIGN_HOME)/$(PLATFORM)/$(DESIGN_NAME)/sram/lib/fakeram_1x256_1r1w.lib
```

Now the final step before we replace the 1x256 `sram_1r1w` module instantiation is to wrap our macro in verilog:

```
module fakeram_1r1w_1x256 (
    clk,
    read_en,
    read_addr,
    read_data,
    write_en,
    write_addr,
    write_data
);
    parameter DATA_WIDTH = 1;
    parameter SIZE = 256;
    parameter READ_DURING_WRITE = "NEW_DATA";
    parameter ADDR_WIDTH = $clog2(SIZE);
    input clk;
    input read_en;
    input [ADDR_WIDTH - 1:0] read_addr;
    output wire [DATA_WIDTH - 1:0] read_data;
    input write_en;
    input [ADDR_WIDTH - 1:0] write_addr;
    input [DATA_WIDTH - 1:0] write_data;
    fakeram_1x256_1r1w sram (
        .r0_clk      (clk),
        .w0_clk      (clk),
        .r0_rd_out  (read_data),
        .r0_addr_in (read_addr),
        .w0_addr_in (write_addr),
        .w0_we_in   (write_en),
        .w0_wd_in   (write_data),
        .r0_ce_in   (read_en),
        .w0_ce_in   (1'b1)
    );
endmodule
```

Take note of the naming conventions for the `fakeram_1x256_1r1w` ports mentioned in our FakeRAM's [README](#) as well as the naming conventions used by **NyuziProcessor** for the `sram_1r1w`

macro. The suite standard is to add the wrapper to a `macros.v` file, which should be located at `designs/src/<design-name>/macros.v`. Furthermore, the file location should also be added under the `VERILOG_FILES` parameter. For more info, [check out NyuziProcessor's full macros.V file.](#)

Now, we can finally replace the original 1x256 `sram_1r1w` instantiation with our FakeRAM:

```
fakeram_1r1w_1x256 #(
    .DATA_WIDTH(1),
    .SIZE(256),
    .READ_DURING_WRITE("NEW_DATA")
) sram_dirty_flags(
    .read_en(l2a_request_valid),
    .read_addr(l2a_request.address.set_idx),
    .read_data(l2t_dirty[way_idx]),
    .write_en(l2r_update_dirty_en[way_idx]),
    .write_addr(l2r_update_dirty_set),
    .write_data(l2r_update_dirty_value),
    /*
);

```

That's it! We've finally replaced an existing SRAM module with FakeRAM!

## Evaluating design quality in ORFS

Once a design reaches completion, we can evaluate it based on power, performance, and area (PPA) metrics. This stage is highly important in the porting process, as we need to ensure that our designs are indicative of modern processor and accelerator microarchitectures. Here's a few things to consider while evaluating a design:

- 1. Less unused (whitespace) is better**
  - Whitespace is defined by **core area - standard cell area**
  - Ideally, we want the core utilization for a design to be at or **above 55%**. This is a common target in modern designs
- 2. Macro placement should look reasonable.**
  - Make sure spacing (and locations) of macros are reasonable on-chip
- 3. It should have an “aggressive” but attainable clock period.**
  - As a rule of thumb, start with a more relaxed timing period and shorten it after each successful run
- 4. Timing constraints should be reasonable (not necessarily the default percentage of the clock period)**
  - Setting output and input delays based on the percentage of the clock period is a very naive approach

- If possible, try to search for timing information regarding the design (most likely found in documentation for a given project/design)
- 5. **If you don't meet timing, look at the STA reports and find the cause.**
  - Is it a setup or hold violation path?
  - Is the clock skew excessive? Can clock buffers be placed between macros?
  - Are the constraints unreasonable or should they be changed?
  - Is the critical setup path going all over the chip?
- 6. **You may need to add custom IO placement or macro placement.**
  - If you notice critical paths from IO pins which route across the entire chip, they should be manually placed
  - If routing fails even with low core utilization, it may be a result of poor macro placement, requiring manual placement
- 7. **It should be routable!**
  - The design needs to pass all DRC checks

## Submitting a Pull Request (PR)

Once your design is good, please submit a PR in GitHub and we can merge your benchmark into our suite!