Prelab 2 - Customizing a Chipyard Generated SoC

This tutorial is divided into two parts plus a Prerequisites section. Part 1 will guide you through writing a new "Config," enabling the creation of a custom SoC by mixing different components generated by Chipyard, such as cores, caches, accelerators, and more. Part 2 will cover integrating a peripheral device (our own RTL written in Verilog) into the Chipyard-generated SoC. Steps on verifying the generated SoC RTL will also be there for both the parts.

Prerequisites

Make sure to complete the steps mentioned in the Lab 1 till **Chipyard Configuration.3**. No need to run the vlsi flow.

https://gitlab.cecs.pdx.edu/west/creating-a-gds/-/blob/main/chipyard_asap7_commercial.md

If you have already done the above steps and have setup chipyard correctly, then run the following two commands. These 2 commands need to be run every time you open a new Xterm. **{Note you can replace $USER with your username}**

1) **exec "$SHELL"**
   This command sources the ~/.bashrc file and sets the environment variables present in the file.
   Make sure to have the following export commands in the .bashrc file, before running the command. Otherwise, copy and paste the missing lines and then run it.

   ```
   export PATH="$PATH":/u/$USER/miniforge3/condabin
   export LANG=en_US.UTF-8
   export PDK_DIR=/stash/asap7/asap7PDK_r1p7
   ```

2) **source env.sh**
   run this command in /u/$USER/creating-a-gds/chipyard
   type **cd ~/creating-a-gds/chipyard/** then type **source env.sh**

Part 1 – Creating a Heterogenous SoC

In Lab 1 we ran the RTL to GDS flow on a TinyRocket Chip.

A TinyRocket Chip is basically a stripped down version of the Rocket Chip.

https://chipyard.readthedocs.io/en/1.10.0/Generators/Rocket-Chip.html

The Chipyard framework mainly makes use of scala programming language. The TinyRocketConfig can be found in the following file (Line 14).

/u/$USER/creating-a-gds/chipyard/generators/chipyard/src/main/scala/config/RocketConfigs.scala

You can use **gvim <above file path>** to open and view/edit the file.

```
1  package chipyard
2
3  import org.chipsalliance.cde.config.{Config}
4  import freechips.rocketchip.diplomacy.{AsynchronousCrossing}
5
6  // --------------
7  // Rocket Configs
8  // --------------
9
10 class RocketConfig extends Config(
11   new freechips.rocketchip.subsystem.WithNBigCores(1) ++         // single rocket-core
12   new chipyard.config.AbstractConfig)
13
14 class TinyRocketConfig extends Config(
15   new chipyard.iobinders.WithDontTouchIOBinders(false) ++         // TODO FIX: Don't dontTouch the ports
16   new freechips.rocketchip.subsystem.WithIncoherentBusTopology ++ // use incoherent bus topology
17   new freechips.rocketchip.subsystem.WithNBanks(0) ++             // remove L2$
18   new freechips.rocketchip.subsystem.WithNoMemPort ++             // remove backing memory
19   new freechips.rocketchip.subsystem.With1TinyCore ++             // single tiny rocket-core
20   new chipyard.config.AbstractConfig)
21
22 class UARTTSIRocketConfig extends Config(
23   new chipyard.harness.WithUARTSerial ++
24   new chipyard.config.WithNoUART ++
25   new chipyard.config.WithMemoryBusFrequency(10) ++
26   new chipyard.config.WithPeripheryBusFrequency(10) ++
27   new freechips.rocketchip.subsystem.WithNBigCores(1) ++         // single rocket-core
28   new chipyard.config.AbstractConfig)
```

Similarly we can create our own configs, create a custom SoC and then run the rest of the vlsi flow on it(synthesis, par, drc & lvs). {Commands provided in Lab 1}

The Chipyard framework involves multiple cores and accelerators that can be composed in arbitrary ways. This discussion will focus on how you combine Rocket, BOOM and Hwacha in particular ways to create a unique SoC.

Note - (Rocket and Boom are cores while Hwacha is an accelerator which can be connected to any of the cores).

To read more about these Chipyard-generated RTL

https://chipyard.readthedocs.io/en/1.10.0/Generators/index.html

Steps to creating a custom SoC.

1) For this tutorial, go to /u/$USER/creating-a-gds/chipyard/generators/chipyard/src/main/scala/config/

2) Open HeteroConfigs.scala

```scala
package chipyard

import org.chipsalliance.cde.config.{Config}

// ---------------------
// Heterogenous Configs
// ---------------------

class LargeBoomAndRocketConfig extends Config(
  new boom.common.WithNLargeBooms(1) ++                          // single-core boom
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++         // single rocket-core
  new chipyard.config.WithSystemBusWidth(128) ++
  new chipyard.config.AbstractConfig)

// DOC include start: BoomAndRocketWithHwacha
class HwachaLargeBoomAndHwachaRocketConfig extends Config(
  new chipyard.config.WithHwachaTest ++
  new hwacha.DefaultHwachaConfig ++                           // add hwacha to all harts
  new boom.common.WithNLargeBooms(1) ++                       // add 1 boom core
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++      // add 1 rocket core
  new chipyard.config.WithSystemBusWidth(128) ++
  new chipyard.config.AbstractConfig)
// DOC include end: BoomAndRocketWithHwacha

class LargeBoomAndHwachaRocketConfig extends Config(
  new chipyard.config.WithMultiRoCC ++                                   // support heterogeneous rocc
  new chipyard.config.WithMultiRoCCHwacha(0) ++                          // put hwacha on hart-0 (rocket)
  new hwacha.DefaultHwachaConfig ++                                      // set default hwacha config keys
  new boom.common.WithNLargeBooms(1) ++                                  // add 1 boom core
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++                 // add 1 rocket core
  new chipyard.config.WithSystemBusWidth(128) ++
  new chipyard.config.AbstractConfig)
```

This scala file includes configs which contains instantiations of more than one type of cores. Similarly we can create a new config according to our liking.

3) Create a new config class and instantiate the AbstractConfig class first. Here, it is called TapeoutClassConfig.

```scala
21   new chipyard.config.WithSystemBusWidth(128) ++
22   new chipyard.config.AbstractConfig)
23 // DOC include end: BoomAndRocketWithHwacha
24
25 class TapeoutClassConfig extends Config(
26
27   new chipyard.config.AbstractConfig
28 )
29
30
31
32 class LargeBoomAndHwachaRocketConfig extends Config(
33   new chipyard.config.WithMultiRoCC ++
34   new chipyard.config.WithMultiRoCCHwacha(0) ++
```

The AbstractConfig class contains the config fragments for constructing the **non-tile** parts of the SoC. Here tile refers to the core and its immediate caches.

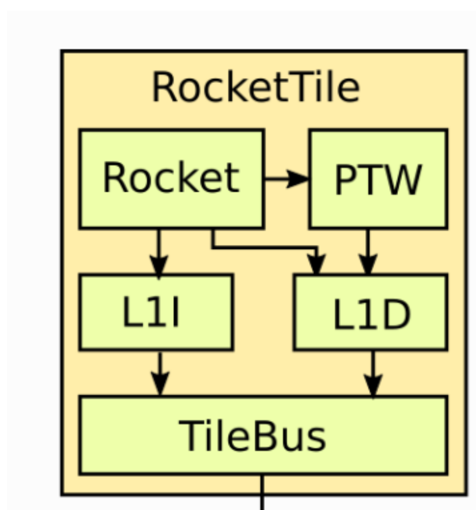

Fig. 1 – An example rocket tile from a rocket chip.

File path: /u/$USER/creating-a-gds/chipyard/generators/chipyard/src/main/scala/config/AbstractConfig.scala

4) Add the rest of the components to our custom config

```
25 class TapeoutClassConfig extends Config(
26   new freechips.rocketchip.subsystem.WithNBigCores(1) ++              // 1 rocket cores added
27   new boom.common.WithNLargeBooms(1) ++
28   new chipyard.config.WithSystemBusWidth(128) ++
29   new chipyard.config.AbstractConfig
30 )
```

Here a simple config is created which contains one boom core and one rocket core.

{Important – Lines 26,27 and 28 are called config fragments. The Config fragments are added from bottom to top. So this way we can override the base values/components mentioned in the AbstractConfig by adding more config fragments on top of it. Not shown in this example as only two different types of cores are added here, which are not there anyway in the AbstractConfig}.

5) Save and close the file.

That's it! A new configuration has been added. Now it's time to generate the RTL for our new SoC and test it. You can play around with the config fragments.

Verification (Bare Metal Testing)

Steps to generate and test the SoC

1) Go to /u/$USER/creating-a-gds/chipyard/tests
This directory contains the C programs for bare metal testing of the generated SoCs.

2) Run **make** command in this directory.
This will invoke the cross compiler to generate riscv-executables, which can be fed to our SoC. You should see a .riscv file for each of the C file, after running make.

3) Now go to /u/$USER/creating-a-gds/chipyard/sims/verilator

4) Run **make CONFIG=TapeoutClassConfig**
This should result in another binary called simulator-chipyard.harness-TapeoutClassConfig.
If that works, it means all the Chisel and Scala are well written, and it's time to check if all the RTL is connected and mapped properly. *{In this example it's easy to write the scala code, so this command should most definitely work. This can be used for checking future designs with more scala coding involved}.*

5) Run **make CONFIG=TapeoutClassConfig BINARY=../../tests/hello.riscv run-binary**

This runs the RISC-V binary of hello.c, where hello.c is a C program which outputs the type of core 0, i.e., whether core 0 is a rocket core or a boom. It is a boom core in this case. You can switch the cores by swapping lines 26 and 27 and then re-running by **make clean** followed by steps 4 and 5 again.

Go to generated-src/chipyard.harness.TestHarness.<project>/gen-collateral to see the generated RTL files.

Part 2 – Integrating a Peripheral Component using MMIO

Chipyard allows us to integrate existing Verilog IPs in the form of black boxes.

Peripheral components can be integrated as a Memory Mapped IO device or as a Tightly coupled Rocc Accelarator (IO device accesed through custom instructions).

In this tutorial we will be going over MMIO integration.

For integrating our Verilog module into a Chipyard SoC, we need to place our Verilog in a specific directory and also write a scala wrapper for proper integration.

Here we will be using the Verilog code for a GCD (Greatest Common Divisor) module, which is already provided with Chipyard. This takes two inputs – x&y and computes the greatest common divisor for the two inputs.

Step 1 – Creating the directory structure

Go to /u/$USER/creating-a-gds/chipyard/generators

Create the following directory structure

```
generators/yourproject/
    build.sbt
    src/main/
        scala/
        resources/
            vsrc/
                YourFile.v
```

Here are the relevant commands to be entered in generators directory. Enter them in the order given.

mkdir jon {you can change jon to anything you wish, but better to keep it as jon for ease of completion of this tutorial.}

cd jon

touch build.sbt

mkdir src

cd src

mkdir main

cd main

mkdir scala

mkdir resources

cd resources

mkdir vsrc

Step 2 – Copy the Verilog and scala code

For this tutorial we are using the Verilog code that is already present in the repo. The scala wrapper is also present the repo. Now we need to copy these 2 files into the corresponding folders in the directory structure we created in step 1.

cp /u/$USER/creating-a-gds/chipyard/generators/chipyard/src/main/scala/example/GCD.scala /u/jonthom/creating-a-gds/chipyard/generators/jon/src/main/scala/

cp /u/jonthom/creating-a-gds/chipyard/generators/chipyard/src/main/resources/vsrc/GCDMMIOBlackBox.v /u/jonthom/creating-a-gds/chipyard/generators/jon/src/main/resources/vsrc/

```
generators/chipyard/
    build.sbt
    src/main/
        scala/
            example/
                GCD.scala
        resources/
            vsrc/
                GCDMMIOBlackBox.v
```

The final directory structure should look like this.

Step 3 – Edit the build.sbt file

We need to add our project to the top level buildfile.

filepath - /u/$USER/creating-a-gds/chipyard/build.sbt

```
230   .settings(chiselTestSettings)
231   .settings(commonSettings)
232
233 lazy val jon = (project in file("generators/jon"))
234   .dependsOn(rocketchip)
235   .settings(libraryDependencies ++= rocketLibDeps.value)
236   .settings(chiselTestSettings)
237   .settings(commonSettings)
238
239
240 lazy val gemmini = (project in file("generators/gemmini"))
241   .dependsOn(rocketchip)
242   .settings(libraryDependencies ++= rocketLibDeps.value)
243   .settings(chiselTestSettings)
244   .settings(commonSettings)
245
```

Here our project name is jon. So add the lines 233-237 as it is in the screenshot above.

Then we need to add our project to the chipyard project in the same buildfile.

```
151 lazy val chipyard = (project in file("generators/chipyard"))
152   .dependsOn(testchipip, rocketchip, boom, hwacha, sifive_blocks, sifive_cache, iocell,
153     sha3, // On separate line to allow for cleaner tutorial-setup patches
154     jon,
155     dsptools, `rocket-dsp-utils`,
156     gemmini, icenet, tracegen, cva6, nvdla, sodor, ibex, fft_generator,
157     constellation, mempress, barf, shuttle)
158   .settings(libraryDependencies ++= rocketLibDeps.value)
159   .settings(
160     libraryDependencies ++= Seq(
161       "org.reflections" % "reflections" % "0.10.2"
162     )
163   )
```

Our project 'jon' has been added to the chipyard dependencies.

After this add our project as a dependency to the tapeout project in the same file

```
259 lazy val tapeout = (project in file("./tools/barstools/"))
260   .dependsOn(jon)  //changed
261   .settings(chiselSettings)
262   .settings(chiselTestSettings)
263   .settings(commonSettings)
```

Step 4 – Edit the DigitalTop.scala file

Before we edit this file it is recommended to change line 1 of the GCD.scala file which we copy-pasted in step 2. In this case the package name is changed to tapeoutclass_package.

```
 1 //package chipyard.example
 2 package tapeoutclass_package
 3
 4 import chisel3._
 5 import chisel3.util._
 6 import chisel3.experimental.{IntParam, BaseModule}
 7 import freechips.rocketchip.amba.axi4._
 8 import freechips.rocketchip.subsystem.BaseSubsystem
 9 import org.chipsalliance.cde.config.{Parameters, Field, Config}
10 import freechips.rocketchip.diplomacy._
11 import freechips.rocketchip.regmapper.{HasRegMap, RegField}
12 import freechips.rocketchip.tilelink._
13 import freechips.rocketchip.util.UIntIsOneOf
14
```

DigitalTop.scala{filepath - /u/$USER/creating-a-gds/chipyard/generators/chipyard/src/main/scala/DigitalTop.scala }

In DigitalTop.scala comment lines 30 and 50 and add our traits(lines 31 and 51) as in the screenshot below. The DigitalTop Module is the actual RTL that gets synthesized.

```
25   with sifive.blocks.devices.gpio.HasPeripheryGPIO // Enables optionally adding the sifive GPIOs
26   with sifive.blocks.devices.spi.HasPeripherySPIFlash // Enables optionally adding the sifive SPI flash controller
27   with sifive.blocks.devices.spi.HasPeripherySPI // Enables optionally adding the sifive SPI port
28   with icenet.CanHavePeripheryIceNIC // Enables optionally adding the IceNIC for FireSim
29   with chipyard.example.CanHavePeripheryInitZero // Enables optionally adding the initzero example widget
30   //with chipyard.example.CanHavePeripheryGCD // Enables optionally adding the GCD example widget
31   with tapeoutclass_package.CanHavePeripheryGCD
32   with chipyard.example.CanHavePeripheryStreamingFIR // Enables optionally adding the DSPTools FIR example widget
33   with chipyard.example.CanHavePeripheryStreamingPassthrough // Enables optionally adding the DSPTools streaming-passth
  rough example widget
34   with nvidia.blocks.dla.CanHavePeripheryNVDLA // Enables optionally having an NVDLA
35   with chipyard.clocking.HasChipyardPRCI // Use Chipyard reset/clock distribution
36   with fftgenerator.CanHavePeripheryFFT // Enables optionally having an MMIO-based FFT block
37   with constellation.soc.CanHaveGlobalNoC // Support instantiating a global NoC interconnect
38 {
39   override lazy val module = new DigitalTopModule(this)
40 }
41
42 class DigitalTopModule[+L <: DigitalTop](l: L) extends ChipyardSystemModule(l)
43   with testchipip.CanHaveTraceIOModuleImp
44   with sifive.blocks.devices.i2c.HasPeripheryI2CModuleImp
45   with sifive.blocks.devices.pwm.HasPeripheryPWMModuleImp
46   with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
47   with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
48   with sifive.blocks.devices.spi.HasPeripherySPIFlashModuleImp
49   with sifive.blocks.devices.spi.HasPeripherySPIModuleImp
50   //with chipyard.example.CanHavePeripheryGCDModuleImp
51   with tapeoutclass_package.CanHavePeripheryGCDModuleImp
52   with freechips.rocketchip.util.DontTouch
53 // DOC include end: DigitalTop
```

Step 5 – Add the new Config

Finally we can add the new config as we did in Part 1. This time we will add our config in MMIOAcceleratorConfigs.scala.

{file path - /u/$USER/creating-a-gds/chipyard/generators/chipyard/src/main/scala/config/MMIOAcceleratorConfigs.scala }

```
10 //Jonathan Config
11 class TapeoutGCD extends Config(
12   new tapeoutclass_package.WithGCD(useAXI4=false, useBlackBox=true) ++         // Use GCD Chisel, connect Tilelink
13   new freechips.rocketchip.subsystem.WithNBigCores(1) ++
14   new chipyard.config.AbstractConfig)
15
```

That's it!

Verification (Bare metal testing)

The RISC-V binaries should already be generated in the test directory if Part 1 of this lab is completed. Else follow steps 1 and 2 in the verification section of Part 1.

1) Go to /u/$USER/creating-a-gds/chipyard/sims/verilator
2) Run **make CONFIG=TapeoutGCD**
3) Run **make CONFIG=TapeoutGCD BINARY=../../tests/gcd.riscv run-binary**

change the x and y input values in gcd.c in the test directory and run make again to generate new gcd.riscv and rerun this step 3 command again to test again.

```
10    while (y != 0) {
11      if (x > y)
12        x = x - y;
13      else
14        y = y - x;
15    }
16    return x;
17 }
18
19 // DOC include start: GCD test
20 int main(void)
21 {
22   uint32_t result, ref, x = 60, y = 40;
23
24   // wait for peripheral to be ready
25   while ((reg_read8(GCD_STATUS) & 0x2) == 0) ;
26
27   reg_write32(GCD_X, x);
28   reg_write32(GCD_Y, y);
29
```

gcd.c {x=60, y=40}

```
[UART] UART0 is here (stdin/stdout).
Hardware result 20 is correct for GCD
- /u/jonthom/creating-a-gds/chipyard/sims/verilator/generated-src/chipyard.harness.TestHarness.TapeoutGCD/gen-collateral/TestDriver.v:158: Verilog $finish
```

terminal output {GCD=20}

i.e., the Greatest Common Divisor of 60 and 40 is 20.

I would highly recommend checking out the following Github repo for clarity regarding the scala/chisel code.

https://github.com/Intensivate/learning-journey/wiki/Adding-an-MMIO-Peripheral