



Generating Heterogeneous Accelerator-rich SoCs in **CHIPYARD**

Jerry Zhao
jzh@berkeley.edu
UC Berkeley



Tutorial Flow

- "Typical" workflow:



- Tutorial workflow:





Outline

5 min – “Integrate” a pre-designed accelerator into the SoC

5 min - Configure a custom shared NoC interconnect

5 min - Generate synthesizable RTL for the SoC

20 min - Compile a RTL simulator of the SoC

5 min - Compile binaries testing accelerators

5 min - Simulate running binaries on the SoC





Chipyard Directory Structure

chipyard-morning/

generators/ ← Our library of Chisel generators

chipyard/

sha3/

sims/ ← Utilities for simulating SoCs

verilator/

firesim/

fpga/ ← Utilities for FPGA prototyping

software/ ← Utilities for building RISC-V software

vlsi/ ← HAMMER VLSI Flow

toolchains/ ← RISC-V Toolchain



Configure the SoC Architecture

Multi-core SoC

- 2 Rocket cores + 1 BOOM core
- 4 banks of L2

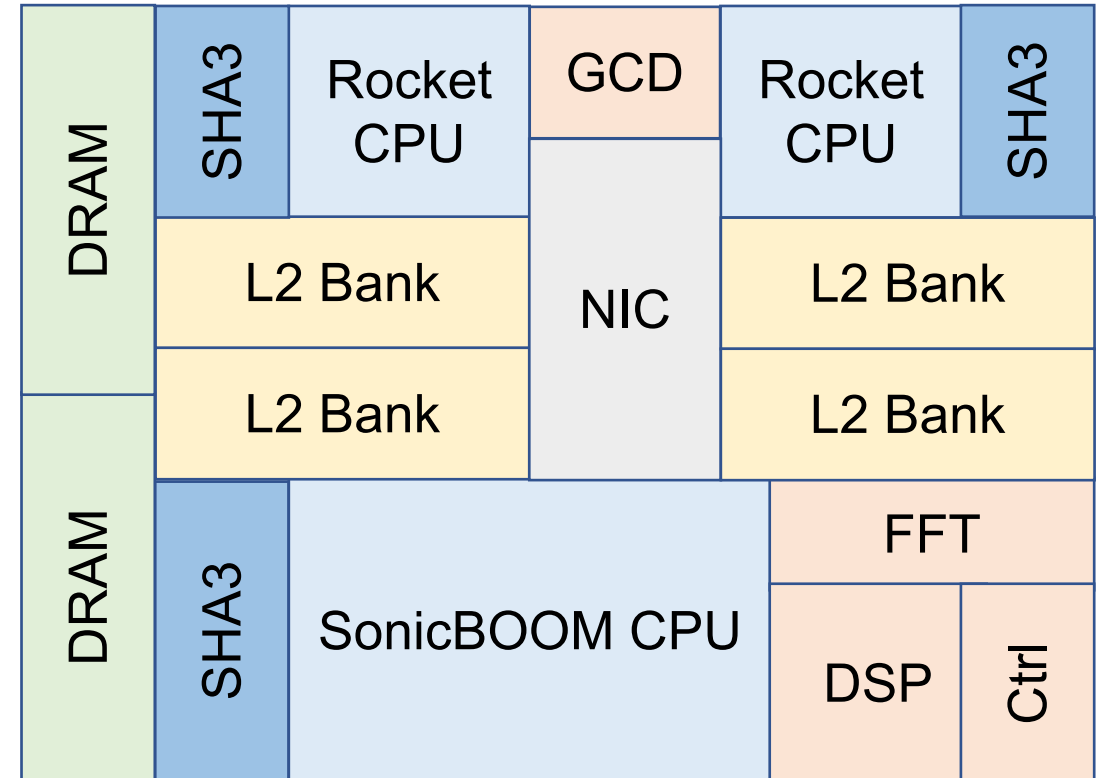
Banked L2/DRAM

Multi-accelerator subsystem

- Streaming FIR “DSP-like” accelerator
- FFT accelerator
- GCD accelerator
- NIC

RoCC tightly-coupled accelerator

- SHA-3





Configure the SoC Architecture

- Open `TutorialConfigs.scala`
- Scroll to the bottom, find `TutorialNoCConfig`
- Scroll to the bottom of `TutorialNoCConfig`

```
// Add custom MMIO devices/accelerators
new chipyard.example.WithGCD ++
new chipyard.harness.WithLoopbackNIC ++
new icenet.WithIceNIC ++
new fftgenerator.WithFFTGenerator(numPoints=8) ++
new chipyard.example.WithStreamingFIR ++
new chipyard.example.WithStreamingPassthrough ++

// Add custom RoCC accelerator
// new sha3.WithSha3Accel ++

// L2 cache configuration
new freechips.rocketchip.subsystem.WithNBanks(4) ++
new freechips.rocketchip.subsystem.WithInclusiveCache(capacityKB = 128) ++

// Core configurations
new boom.common.WithNSmallBooms(1) ++
new freechips.rocketchip.subsystem.WithNBigCores(2) ++

// Inherit default configs
new chipyard.config.AbstractConfig
```

```
chipyard-morning/
  generators/
    chipyard/
      src/main/scala/config/
        TutorialConfigs.scala
```



Configure the SoC Architecture

- Uncomment `WithSha3Accel`

```
// Add custom RoCC accelerator  
new sha3.WithSha3Accel ++
```

Uncomment

```
chipyard-morning/  
  generators/  
    chipyard/  
      src/main/scala/config/  
        TutorialConfigs.scala
```



Configure the NoC Architecture

- Find definitions of `inNodeMapping` and `outNodeMapping`

```
inNodeMapping = ListMap("Core" -> 7),
outNodeMapping = ListMap(
  "pbus" -> 8, "uart" -> 9, "control" -> 10, "gcd" -> 11,
  "writeQueue[0]" -> 0, "writeQueue[1]" -> 1, "tailChain[0]" -> 2))
```

```
inNodeMapping = ListMap(
  "Core 0" -> 0, "SHA3[0]" -> 0,
  "Core 1" -> 1, "SHA3[1]" -> 1,
  "Core 2" -> 2, "SHA3[2]" -> 2,
  "serial-t1" -> 2),
outNodeMapping = ListMap(
  "system[0]" -> 3, "system[1]" -> 4, "system[2]" -> 5, "system[3]" -> 6,
  "pbus" -> 7))
```

```
chipyard-morning/
  generators/
    chipyard/
      src/main/scala/config/
        TutorialConfigs.scala
```




Configure the NoC Architecture

- Modify mapping target nodes to the range [0, 12)
- Don't think too hard about where things go

```
chipyard-morning/  
generators/  
chipyard/  
src/main/scala/config/  
TutorialConfigs.scala
```

```
inNodeMapping = ListMap("Core" -> 7),  
outNodeMapping = ListMap(  
  "pbus" -> 8, "uart" -> 9, "control" -> 10, "gcd" -> 11,  
  "writeQueue[0]" -> 0, "writeQueue[1]" -> 1, "tailChain[0]" -> 2))  
  
inNodeMapping = ListMap(  
  "Core 0" -> 0, "SHA3[0]" -> 0,  
  "Core 1" -> 1, "SHA3[1]" -> 1,  
  "Core 2" -> 2, "SHA3[2]" -> 2,  
  "serial-tl" -> 2),  
outNodeMapping = ListMap(  
  "system[0]" -> 3, "system[1]" -> 4, "system[2]" -> 5, "system[3]" -> 6,  
  "pbus" -> 7))
```



Compile a RTL Simulator

- Go to: `~/chipyard-morning/sims/verilator`
- Run `make CONFIG=TutorialNoCConfig -j16`
- Don't forget **-j16!**
- This will take ~20 minutes

- I'll explain what's happening while the RTL is building

```
chipyard-morning/  
generators/  
sims/  
  verilator/
```



How Configs Work

- A **Config** maintains a dictionary of keys describing how to generate different SoC components
- **Config fragments** set keys or groups of keys
- **Configs** are compositions of **fragments**

```
// Add custom MMIO devices/accelerators
new chipyard.example.WithGCD ++
new chipyard.harness.WithLoopbackNIC ++
new icenet.WithIceNIC ++
new fftgenerator.WithFFTGenerator(numPoints=8) ++
new chipyard.example.WithStreamingFIR ++
new chipyard.example.WithStreamingPassthrough ++

// Add custom RoCC accelerator
// new sha3.WithSha3Accel ++

// L2 cache configuration
new freechips.rocketchip.subsystem.WithNBanks(4) ++
new freechips.rocketchip.subsystem.WithInclusiveCache(capacityKB = 128) ++

// Core configurations
new boom.common.WithNSmallBooms(1) ++
new freechips.rocketchip.subsystem.WithNBigCores(2) ++

// Inherit default configs
new chipyard.config.AbstractConfig
```



Example of a Config Fragment

```
class WithSha3Accel extends Config ((site, here, up) => {  
  case Sha3WidthP => 64  
  case Sha3Stages => 1  
  case Sha3FastMem => true  
  case Sha3BufferSram => false  
  case BuildRoCC => Seq(  
    (p: Parameters) => {  
      val sha3 = LazyModule.apply(  
        new Sha3Accel(OpcodeSet.custom2)(p)  
      )  
      sha3  
    }  
  )  
})
```

SHA3 Parameters

Rocket Chip uses the "BuildRoCC" key to figure out which accelerator to build



SoC Architecture Config

Custom config-specific fragments

```
// Add custom MMIO devices/accelerators
new chipyard.example.WithGCD ++
new chipyard.harness.WithLoopbackNIC ++
new icenet.WithIceNIC ++
new fftgenerator.WithFFTGenerator(numPoints=8) ++
new chipyard.example.WithStreamingFIR ++
new chipyard.example.WithStreamingPassthrough ++

// Add custom RoCC accelerator
// new sha3.WithSha3Accel ++

// L2 cache configuration
new freechips.rocketchip.subsystem.WithNBanks(4) ++
new freechips.rocketchip.subsystem.WithInclusiveCache(capacityKB = 128) ++

// Core configurations
new boom.common.WithNSmallBooms(1) ++
new freechips.rocketchip.subsystem.WithNBigCores(2) ++

// Inherit default configs
new chipyard.config.AbstractConfig
```

Default settings



Default Config

```
// The HarnessBinders control generation of hardware in the TestHarness
new chipyard.harness.WithUARTAdapter ++ // add UART adapter to display UART on stdout, if uart is present
new chipyard.harness.WithBlackBoxSimMem ++ // add SimDRAM DRAM model for axi4 backing memory, if axi4 mem is enabled
new chipyard.harness.WithSimSerial ++ // add external serial-adapter and RAM
new chipyard.harness.WithSimDebug ++ // add SimJTAG or SimDTM adapters if debug module is enabled
new chipyard.harness.WithGPIOtiedOff ++ // tie-off chiptop GPIOs, if GPIOs are present
new chipyard.harness.WithSimSPIFlashModel ++ // add simulated SPI flash memory, if SPI is enabled
new chipyard.harness.WithSimAXIMMIO ++ // add SimAXIMem for axi4 mmio port, if enabled
new chipyard.harness.WithTieOffInterrupts ++ // tie-off interrupt ports, if present
new chipyard.harness.WithTieOffL2FBusAXI ++ // tie-off external AXI4 master, if present
new chipyard.harness.WithTieOffCustomBootPin ++
```

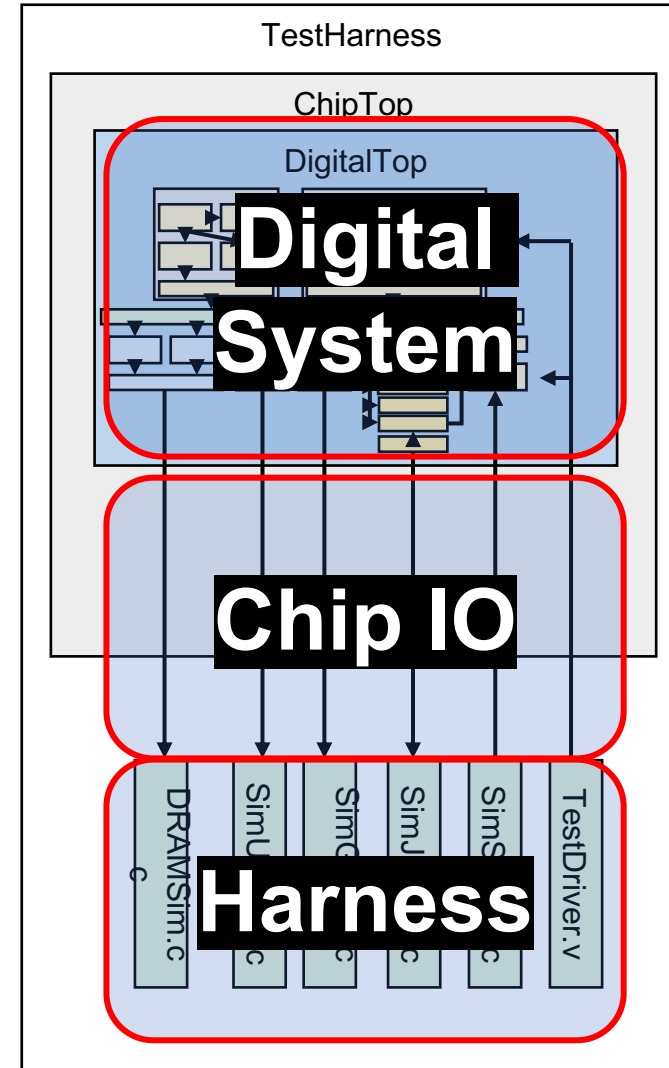
Configuring RTL in the TestHarness

```
// The IObinders instantiate chipyard IOs to match desired digital IOs
// IOCells are generated for "Chip-like" IOs, while simulation-only IOs are directly punched through
new chipyard.iobinders.WithAXI4MemPunchthrough ++
new chipyard.iobinders.WithAXI4MMIOPunchthrough ++
new chipyard.iobinders.WithL2FBusAXI4Punchthrough ++
new chipyard.iobinders.WithBlockDeviceIOPunchthrough ++
new chipyard.iobinders.WithNICIOPunchthrough ++
new chipyard.iobinders.WithSerialTLIOCells ++
new chipyard.iobinders.WithDebugIOCells ++
new chipyard.iobinders.WithUARTIOCells ++
new chipyard.iobinders.WithGPIOCells ++
new chipyard.iobinders.WithUARTIOCells ++
new chipyard.iobinders.WithSPIOCells ++
new chipyard.iobinders.WithTraceIOPunchthrough ++
new chipyard.iobinders.WithExtInterruptIOCells ++
new chipyard.iobinders.WithCustomBootPin ++
```

Configuring IOs in the ChipTop

```
new testchipip.WithDefaultSerialTL ++ // use serialized tilelink port to external serialadapter/harnessRAM
new chipyard.config.WithBootROM ++ // use default bootrom
new chipyard.config.WithUART ++ // add a UART
new chipyard.config.WithL2TLBs(1024) ++ // use L2 TLBs
new chipyard.config.WithNoSubsystemDrivenClocks ++ // drive the subsystem diplomatic clocks from ChipTop instead of using im
new chipyard.config.WithInheritBusFrequencyAssignments ++ // Unspecified clocks within a bus will receive the bus frequency if set
new chipyard.config.WithPeripheryBusFrequencyAsDefault ++ // Unspecified frequencies with match the pbus frequency (which is always s
new chipyard.config.WithMemoryBusFrequency(100.0) ++ // Default 100 MHz mbus
new chipyard.config.WithPeripheryBusFrequency(100.0) ++ // Default 100 MHz pbus
new freechips.rocketchip.subsystem.WithJtagDTM ++ // set the debug module to expose a JTAG port
new freechips.rocketchip.subsystem.WithNoMMIOPort ++ // no top-level MMIO master port (overrides default set in rocketchip)
new freechips.rocketchip.subsystem.WithNoSlavePort ++ // no top-level MMIO slave port (overrides default set in rocketchip)
new freechips.rocketchip.subsystem.WithInclusiveCache ++ // use Sifive L2 cache
new freechips.rocketchip.subsystem.WithNextTopInterrupts(0) ++ // no external interrupts
new chipyard.WithMultiLockCoherentBusTopology ++ // hierarchical buses including mbus+l2
new freechips.rocketchip.system.BaseConfig) // "base" rocketchip system
```

Configuring DigitalSystem components





NoC Config

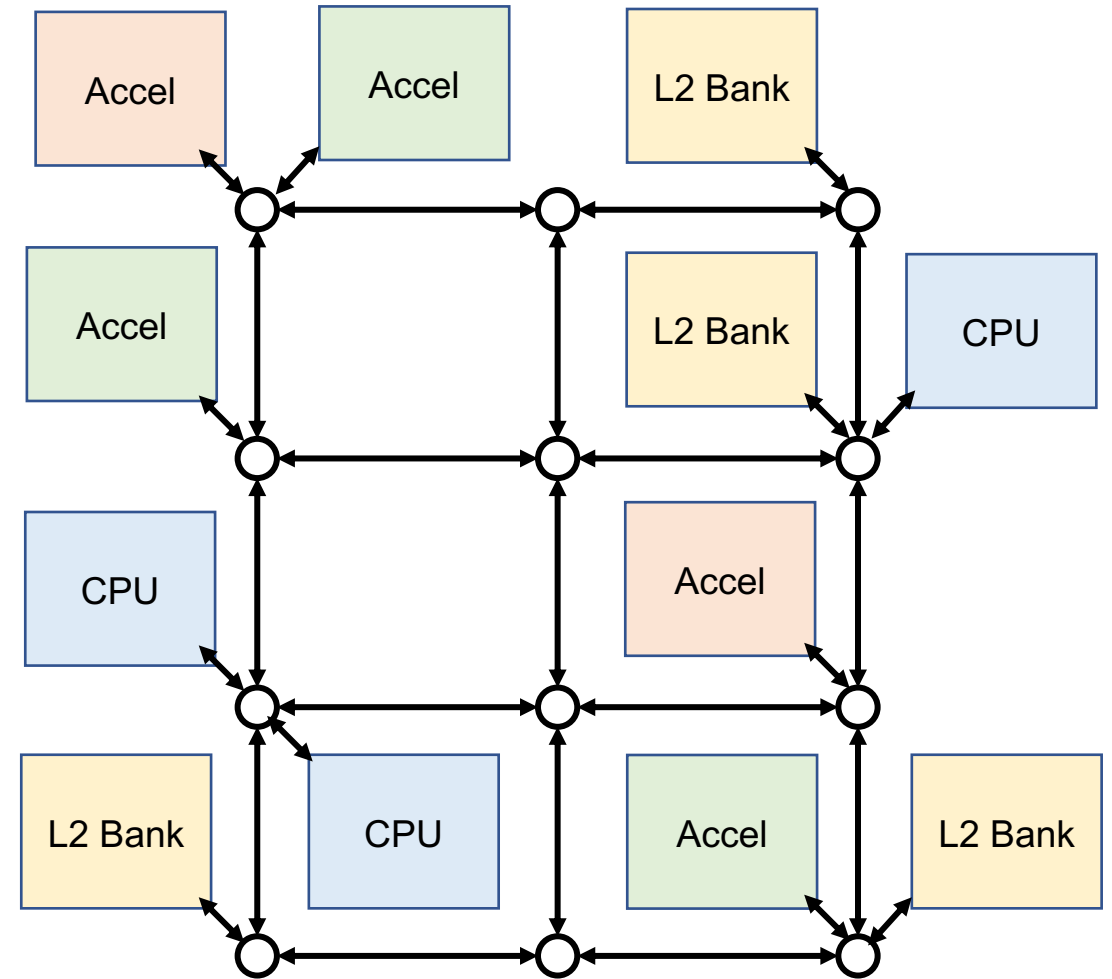
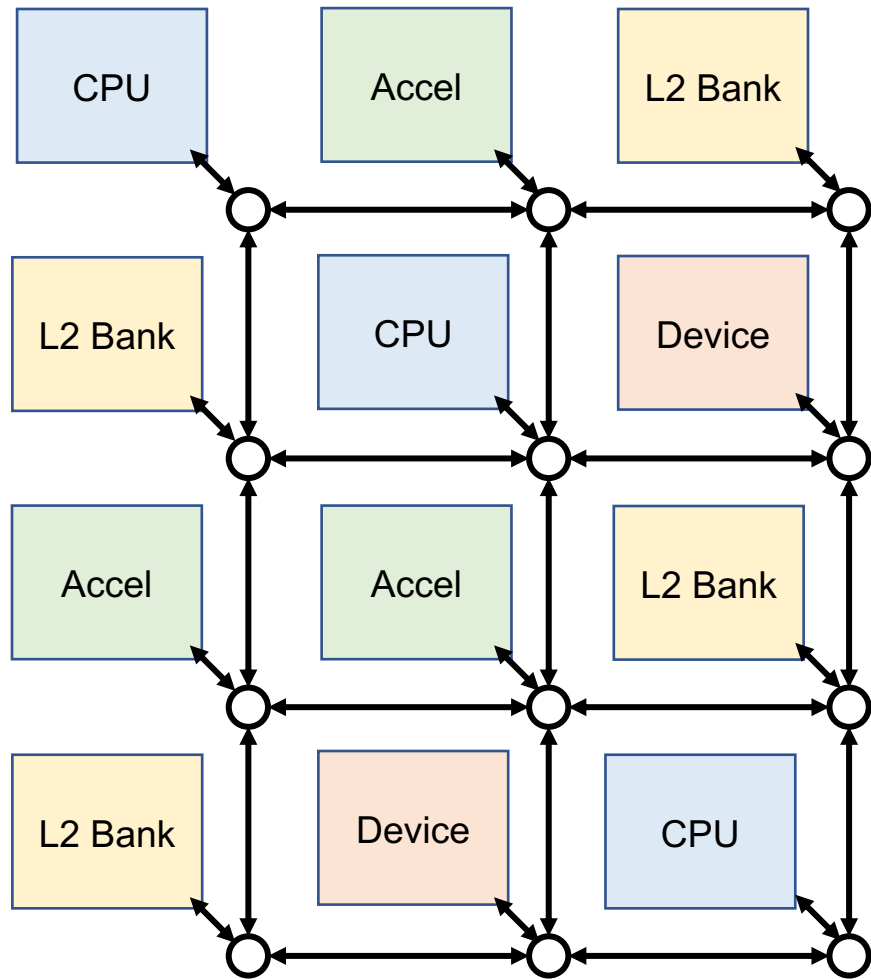
Physical NoC parameters

Mapping agents to Node IDs

```
// Try changing the dimensions of the Mesh topology
new constellation.soc.WithGlobalNoC(constellation.soc.GlobalNoCParams(
  NoCParams(
    topology      = TerminalRouter(Mesh2D(3, 4)),
    channelParamGen = (a, b) => UserChannelParams(Seq.fill(12) { UserVirtualChannelParams(4) }),
    routingRelation = NonblockingVirtualSubnetworksRouting(TerminalRouterRouting(
      Mesh2DEscapeRouting()), 10, 1)
  )
)) ++
// The inNodeMapping and outNodeMapping values are the physical identifiers of
// routers on the topology to map the agents to. Try changing these to any
// value within the range [0, topology.nNodes)
new constellation.soc.WithPbusNoC(constellation.protocol.TLNoCParams(
  constellation.protocol.DiplomaticNetworkNodeMapping(
    inNodeMapping = ListMap("Core" -> 7),
    outNodeMapping = ListMap(
      "pbus" -> 8, "uart" -> 9, "control" -> 10, "gcd" -> 11,
      "writeQueue[0]" -> 0, "writeQueue[1]" -> 1, "tailChain[0]" -> 2))
), true) ++
new constellation.soc.WithSbusNoC(constellation.protocol.TLNoCParams(
  constellation.protocol.DiplomaticNetworkNodeMapping(
    inNodeMapping = ListMap(
      "Core 0" -> 0, "SHA3[0]" -> 0,
      "Core 1" -> 1, "SHA3[1]" -> 1,
      "Core 2" -> 2, "SHA3[2]" -> 2,
      "serial-t1" -> 2),
    outNodeMapping = ListMap(
      "system[0]" -> 3, "system[1]" -> 4, "system[2]" -> 5, "system[3]" -> 6,
      "pbus" -> 7))
), true) ++
```



NoC Mapping





Check Verilog

- Go into the **verilator/generated-src** directory
- Find the directory named **TutorialNoCConfig**
- **ls** the files inside

- **XXX.top.v**: Synthesizable Verilog source
- **XXX.harness.v**: TestHarness
- **XXX.dts**: device tree string
- **XXX.memmap.json**: memory map

```
chipyard-morning/  
  generators/  
    sims/verilator/  
      generated-src/  
        chipyard...TutorialNoCConfig/
```



Compile MMIO Tests

- `cd ~/chipyard-morning/tests`
- Run `make`

- Afterwards, see the `.riscv` bare-metal binaries compiled here

- `gcd.riscv`
- `fft.riscv`
- `nic-loopback.riscv`
- `streaming-fir.riscv`

```
chipyard-morning/  
  generators/  
    tests/
```



Run Test

- Go back to `sims/verilator`
- `make CONFIG=TutorialNoCConfig \`
`run-binary-hex \`
`BINARY=../../tests/fft.riscv`
 - This is a one-line command
- The simulation should print some output, saying test passed successfully
- Try:
 - `fft.riscv`
 - `gcd.riscv`
 - `streaming-fir.riscv`
 - `nic-loopback.riscv`

```
chipyard-morning/  
generators/  
sims/verilator/
```



Compile SHA3 Tests

```
printf("Start basic test 1.\n");  
// BASIC TEST 1 - 150 zero bytes  
  
start = rdcycle();  
  
asm volatile ("fence");  
  
// setup accelerator with addresses of input and output  
ROCC_INSTRUCTION_SS(2, &input, &output, 0);  
  
// Set length and compute hash  
ROCC_INSTRUCTION_S(2, sizeof(input), 1);  
  
asm volatile ("fence" ::: "memory");  
  
end = rdcycle();
```

```
chipyard-morning/  
  generators/  
    sha3/  
      software/  
        tests/  
          src/  
            sha3-rocc.c
```



Compile SHA3 Tests

```
cd ~/chipyard-morning/generators/sha3/software  
./build.sh
```

```
chipyard-morning/  
  generators/  
    sha3/  
      software/  
        build.sh
```

- What was done... built two binaries
 - `sha3-sw.riscv` - software version of SHA3 computation
 - `sha3-rocc.riscv` - sends SHA3 computation to the accelerator
- Both binaries created in
 - `sha3/software/tests/bare/sha3-*.riscv`



Run SHA3 Tests

- Go back to sims/verilator
- make `CONFIG=TutorialNoCConfig \`
`run-binary-hex \`
`BINARY=$SHA3SW/sha3-rocc.riscv`
- The simulation should print some output, saying test passed successfully
- Compare sha3-rocc.riscv with sha3-sw.riscv

```
chipyard-morning/  
generators/  
sims/verilator/
```



Try things on your own

- Change number of cores
 - `WithNBigCores`
 - Uncomment `WithNSmallBoomCores`
- Change number/capacity of L2 banks
 - `WithNBanks`
 - `WithInclusiveCache`
- Change NoC topology
 - Uncomment the block to switch to a heterogeneous irregular NoC
 - Change dimensions of mesh
- Add/remove peripheral accelerators